# UNIT I

object oriented programming

# Introduction to Object-Oriented Programming

- Object-Oriented Programming (OOP) is the term used to describe a programming approach based on objects and classes. The object-oriented paradigm allows us to organise software as a collection of objects that consist of both data and behaviour. This is in contrast to conventional functional programming practice that only loosely connects data and behaviour.

# The object-oriented programming approach encourages:

- Modularisation: where the application can be decomposed into modules.
- Software re-use: where an application can be composed from existing and new modules.

# What is an Object-Oriented Programming Language?

- Computers are powerful machines. With a computer, we can calculate numbers extremely quickly, and we are able to produce amazing programs for many applications. However, to take advantage of this power, we need to communicate with the computer is something less painful than manually typing ones and zeros.
- Therefore, we have programming languages, which are propped up by machine code that's already been written. However, the further removed we get from machine code the more abstract and specialized languages become in handling data. This is why we have so many different languages; no one language is perfect, and they all have different and overlapping applications

# Object-Oriented Programming

- Object-Oriented programming is built around objects, data structures that contain both data (properties or attributes) and code (procedures or methods). Objects are able to modify themselves (with ‘this’ or ‘self’) and in most OOP languages almost everything is an object that can have both values and executable code. Each object is unique, and though it may be a copy of another object it’s variables can be different from any other object’s variables.
- The objects in object-oriented software design can be thought of like actual objects. Think of an object, like a watch. That watch has properties, it’s made of metal, it’s black in color, it has a specific weight. But that object also does things, it displays the time, and it also can affect itself; spinning gears to change the position of its hands.

# Object-Oriented Programming

- Another feature of objects is that we don’t always need to know how the watch functions to get it to function. Assuming the watch is well built it will reliably tell time, without us having to interfere with its inner workings.
- Object oriented languages have objects similar to real world objects. They can have properties and functions. They also tend to follow a certain set of principles.

# The Pros and Cons of Object-Oriented Programming Languages

**Pros-**

**Reusability** – Object-oriented code is extremely modular by design, and because of polymorphism and abstraction you can make one function that can be used over and over again, or copy information and functionality that's already been written with inheritance. This saves time, reduces complexity, saves space, and makes coding a lighter load on our fingers.

**Parallel Development** – because of the four pillars, when the main classes are defined there is enough groundwork for parts of the program to be developed separately from each other and still function. This makes concurrent development much easier for larger development teams.

**Maintenance** – Because most, if not all, of our code is in one place, being called and reused, that code is much easier to maintain. Instead of having to go through a hundred different instances where a function is called and fix each individually, we can fix the one modular and polymorphic function that's called a hundred times.

**Security** – While most languages have some security, object-oriented languages are convenient because security is built-in with encapsulation. Other methods and classes cannot access private data by default, and programs written in OOP languages are more secure for it.

# Pros-

- Reflects the real world – Because objects in object-oriented programming act like real-life objects, the code is much easier to use and visualize. We think about the world around us this way–of objects having information and functions—so programming this way is a little less challenging.

# Cons

- **Often Messy** – Because object-oriented languages are so customizable and scalable, it can be easy to lose an understanding of how the code works. OOP code can function in many ways, and there are many methodologies for programming in OOP that don't work well with other methodologies, or are just inefficient or difficult to use to begin with. Essentially, as programmers are often allowed too much freedom, that freedom is often abused to make programs that could be much more efficient, or are inherently buggy.
- **Requires More Planning** – Because these languages are so modular and scalable, going in without a clear design ahead of time is a recipe for disaster. Creating an efficient program requires a solid plan, more so than with other programming paradigms. Programmers need to have a good understanding of what they're doing to make a program that can be as scalable, efficient, and powerful as the object-oriented language will allow it to be.

# Cons

- **Opacity** – This is as much a pro as it is a con. Because objects and functions can operate so independently, taking in information and (usually) spitting out reliable results, they can end up being black boxes—where what they do isn't always apparent. While the programmer probably created that object and knows what it does, OOP languages simply aren't as transparent as other languages. In functional languages, all of the code is laid out in front of them instead of being parceled up and hidden away like in object-oriented languages.
- **Performance** – Object-oriented languages often take a performance hit. Programs made in OOP languages are often larger in size and require more computational effort to run than functional languages. However, this isn't always true, or important. C++ is an OOP language, but it's one of the fastest languages available. On the same note, speed isn't always important, and the difference in speed really only becomes apparent when processing huge or complex calculations, or in instances where extreme speed is required. Otherwise, most programming languages will function at an apparent same speed depending on the function.

# Popular Object-Oriented Languages

- **Java** – Java is everywhere, and it is one of the most used and in-demand languages of all time. Java's motto is 'write once, run anywhere,' and that is reflected in the number of platforms it runs on, and places it's used.

public static void main(String args[]){System.out.printls("hello");}

- **Python** – Python is general purpose, and used in many places. However, Python has a strong foothold in the machine learning and data science and is one of the preferred languages for that new and ever-growing field.print('hello');
- **C++** – C++ has the speed of C with the functionality of classes and an object-oriented paradigm. It's a compiled, fast, reliable, and powerful language used in so many applications that it's even used to build compilers and interpreters for other languages.

  cout<<"hello";

# CHARACTERSTICS OF OOPS

- Emphasis on data rather than procedure.
- Programs are divided into small parts called object.
- Data structures are designed such that they characterized the objects.
- Data and related functions stay tied in classes.
- Data is secured as they can’t be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be added easily whenever necessary.
- Follows a bottom-up approach in program design.
- Using inheritance, a class can be modified to create another one.
- If necessary, system size can also be modified easily.
- Multiple Programmer System Design is easy using OOP concept.
- The complexity can be reduced.
- Data Type can be created based on the necessity.

# APPLICATIONS OF OOPS

1. **Simulations and Modeling:** Simulation is the technique of representing the real world entities with the help of a computer program. Simula-67 and Smalltalk are two object-oriented languages are designed for making simulations.

2. **User-interface design:** Another popular application of OOP has been in the area of designing graphical user interfaces such as Windows. C++ is mainly used for developing user-interfaces.

3. **Developing computer games**: OOP is also used for developing computer games such as Diablo, Startcraft and Warcraft III. These games offer virtual reality environments in which a number of objects interact with each other in complex ways to give the desired result.

4. **Scripting**: In recent years, OOP has also been used for developing HTML, XHTML and XML documents for the Internet. Python, Ruby and Java are the scripting languages based on object-oriented principles which are used for scripting.

5. **Object Databases:** These days OOP concepts have also been introduced in database systems to develop a new DBMS named object databases. These databases store the data directly in the form of objects. However, these databases are not as popular as the traditional RDBMS.

# Difference between Procedural Programming and Object Oriented Programming:

| PROCEDURAL ORIENTED PROGRAMMING | OBJECT ORIENTED PROGRAMMING |
|---|---|
| In procedural programming, program is divided into small parts called functions. | In object oriented programming, program is divided into small parts called objects. |
| Procedural programming follows top down approach. | Object oriented programming follows bottom up approach. |
| There is no access specifier in procedural programming. | Object oriented programming have access specifiers like private, public, protected etc. |
| Adding new data and function is not easy. | Adding new data and function is easy. |

| PROCEDURAL ORIENTED PROGRAMMING | OBJECT ORIENTED PROGRAMMING |
| --- | --- |
| Procedural programming does not have any proper way for hiding data so it is less secure. | Object oriented programming provides data hiding so it is more secure. |
| In procedural programming, overloading is not possible. | Overloading is possible in object oriented programming. |
| In procedural programming, function is more important than data. | In object oriented programming, data is more important than function. |
| Procedural programming is based on unreal world. | Object oriented programming is based on real world. |
| Examples: C, FORTRAN, Pascal, Basic etc. | Examples: C++, Java, Python, C# etc. |

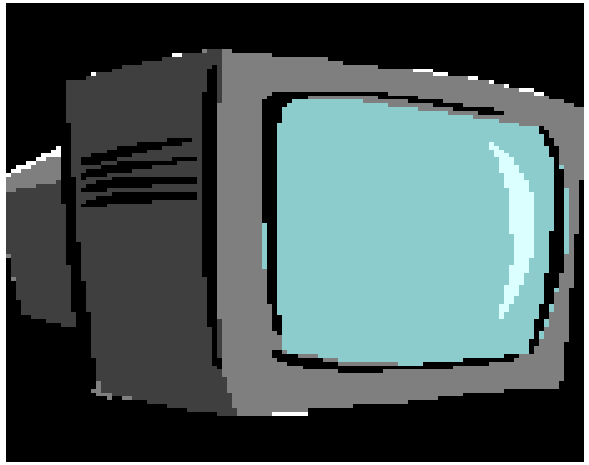
# An object-oriented programming language generally supports five main features:

- Classes
- Objects
- Classification
- Polymorphism
- Inheritance

# An Object-Oriented Class

- **If we think of a real-world object, such as a television it will have several features and properties:**
- We do not have to open the case to use it.
- We have some controls to use it (buttons on the box, or a remote control).
- We can still understand the concept of a television, even if it is connected to a DVD player.
- It is complete when we purchase it, with any external requirements well documented.
- The TV will not crash!

# The concept of a class - television example.



**A class should:**

Provide a well-defined interface - such as the remote control of the television.

Represent a clear concept - such as the concept of a television.

Be complete and well-documented - the television should have a plug and should have a manual that documents all features.

The code should be robust - it should not crash, like the television.

# CLASS

With a functional programming language (like C) we would have the component parts of the television scattered everywhere and we would be responsible for making them work correctly - there would be no case surrounding the electronic components.

Humans use class based descriptions all the time - what is a duck? (Think about this, we will discuss it soon.)

Classes allow us a way to represent complex structures within a programming language. They have two components:

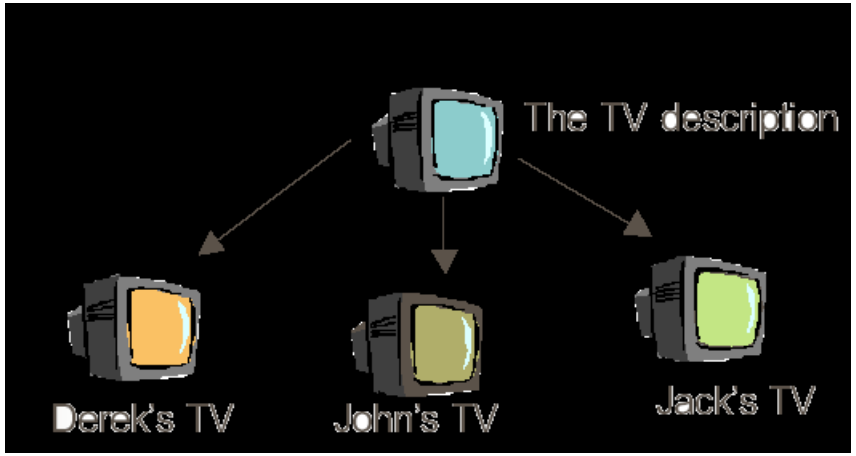
- **States -** (or data) are the values that the object has.
- **Methods** - (or behaviour) are the ways in which the object can interact with its data, the actions.

# An instance of a class is called an **object.**

- **An Object**

An object is an instance of a class. You could think of a class as the description of a concept, and an object as the realization of this description to create an independent distinguishable entity. For example, in the case of the Television, the class is the set of plans (or blueprints) for a generic television, whereas a television object is the realization of these plans into a real-world physical television. So there would be one set of plans (the class), but there could be thousands of real-world televisions (objects).

# The Television objects example.



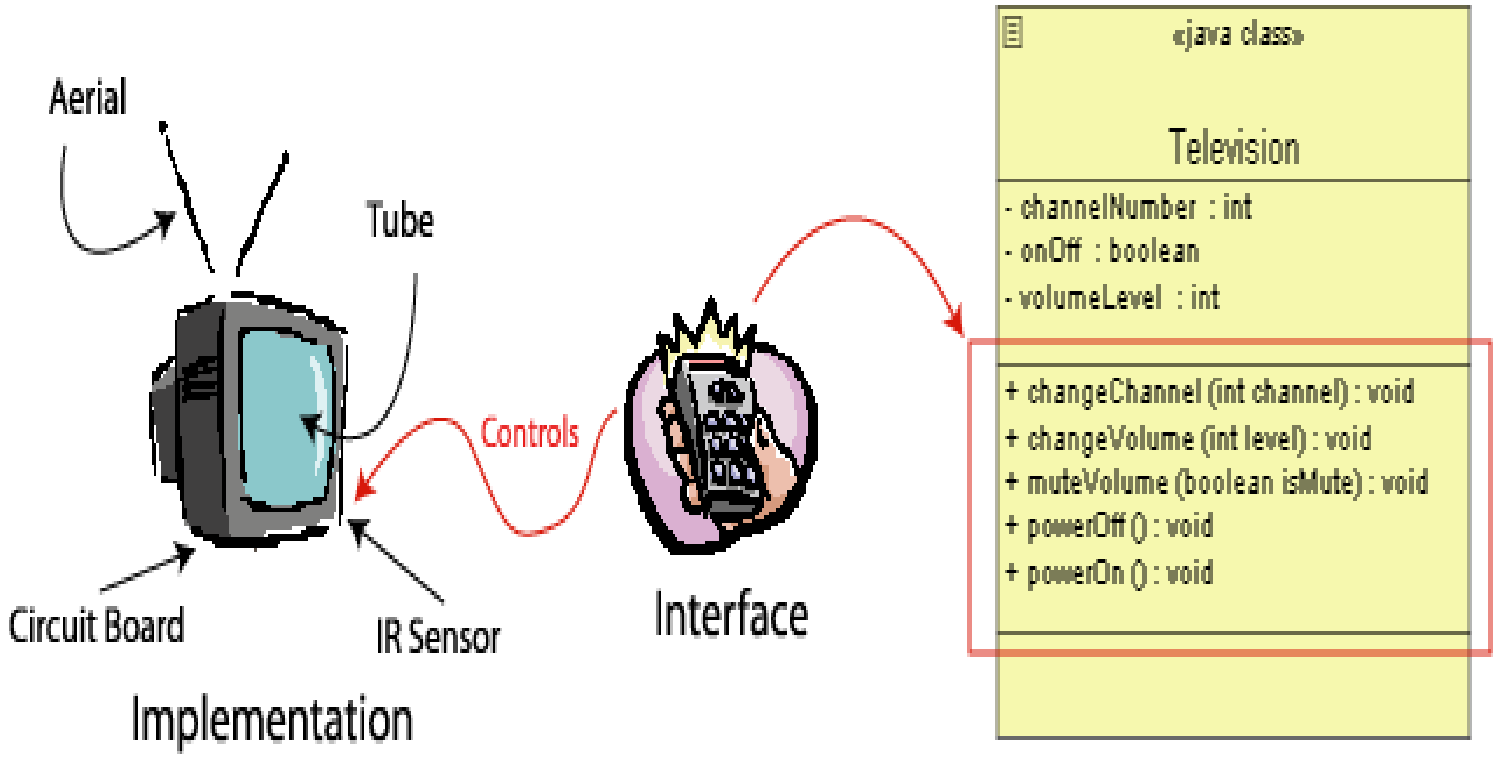
# Encapsulation

- The object-oriented paradigm encourages encapsulation. Encapsulation is used to hide the mechanics of the object, allowing the actual implementation of the object to be hidden, so that we don't need to understand how the object works. All we need to understand is the interface that is provided for us.

- You can think of this in the case of the Television class, where the functionality of the television is hidden from us, but we are provided with a remote control, or set of controls for interacting with the television, providing a high level of abstraction. So, as in Figure 1.4 there is no requirement to understand how the signal is decoded from the aerial and converted into a picture to be displayed on the screen before you can use the television.

- There is a sub-set of functionality that the user is allowed to call, termed the interface. In the case of the television, this would be the functionality that we could use through the remote control or buttons on the front of the television.

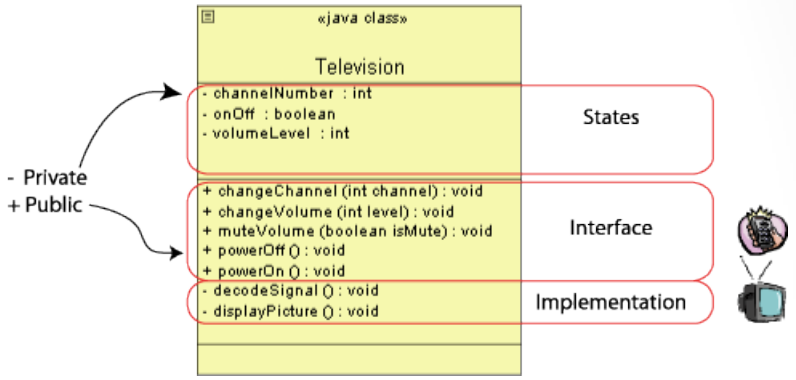
# The Television interface example.



# Encapsulation

Encapsulation is the term used to describe the way that the interface is separated from the implementation. You can think of encapsulation as "data-hiding", allowing certain parts of an object to be visible, while other parts remain hidden. This has advantages for both the user and the programmer.

lass example showing encapsulation.




The Television class example showing encapsulation.

# Class and Object

- The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

- An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

# Class and Object

```
class person
{
    char name[20];
    int id;
public:
    void getdetails(){}
};

int main()
{
   person p1; // p1 is a object
}
```

# Encapsulation

In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

# Abstraction

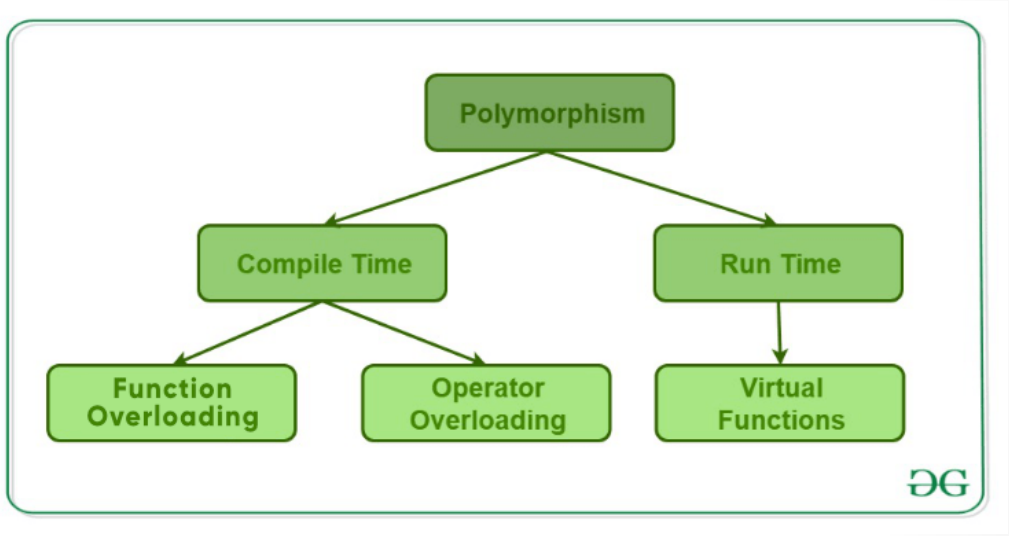
- Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the 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.

# 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. A real-life example 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 posses different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

# In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism



# Compile time polymorphism:

- This type of polymorphism is achieved by function overloading or operator overloading.

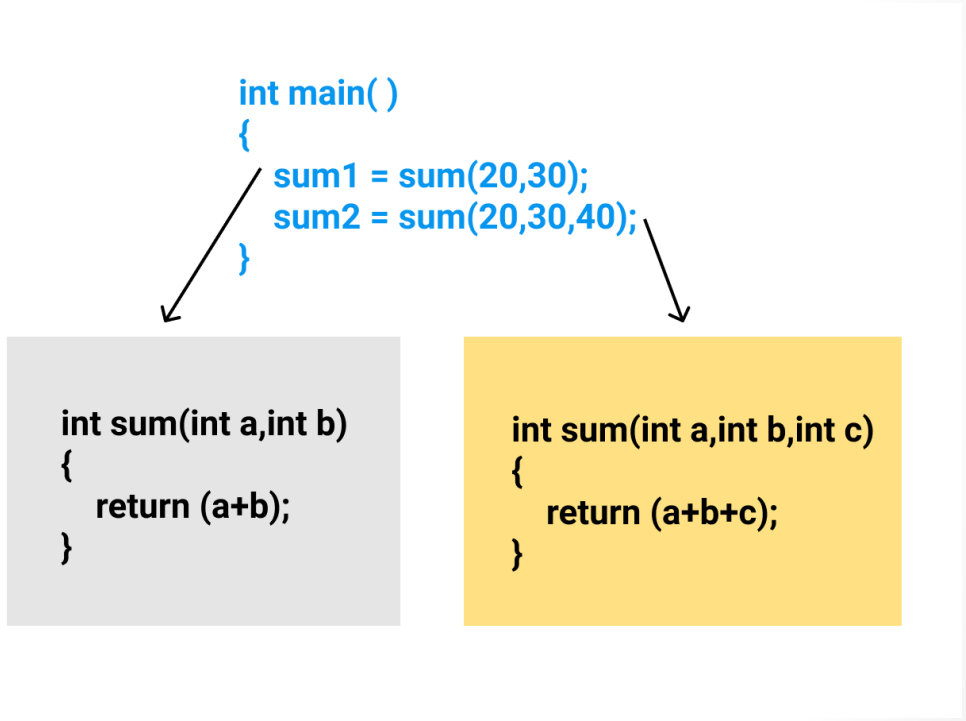
add (int x, int y)

add (String x, String y)

When there are multiple functions with same name but different parameters then these **functions are said to be overloaded.** Functions can be overloaded by change in number of arguments or/and change in type of arguments.

# Function overloading.

- **Example**: Suppose we have to write a function to add some integers, some times there are 2 integers, some times 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.



# Access Modifiers in C++

- Access modifiers are used to implement an important feature of Object-Oriented Programming known as Data Hiding. Consider a real-life example:

- The Indian secret informatic system having 10 senior members have some top secret regarding national security. So we can think that 10 people as class data members or member functions who can directly access secret information from each other but anyone can't access this information other than these 10 members i.e. outside people can't access information directly without having any privileges. This is what data hiding is.

There are 3 types of access modifiers available in C++:

- Public
- Private
- Protected

# Let us now look at each one these access modifiers in details:

- 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.

```
public:
    double radius;

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

# 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 are allowed to access the private data members of a class.

```
private:
        double radius;
```

# 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.

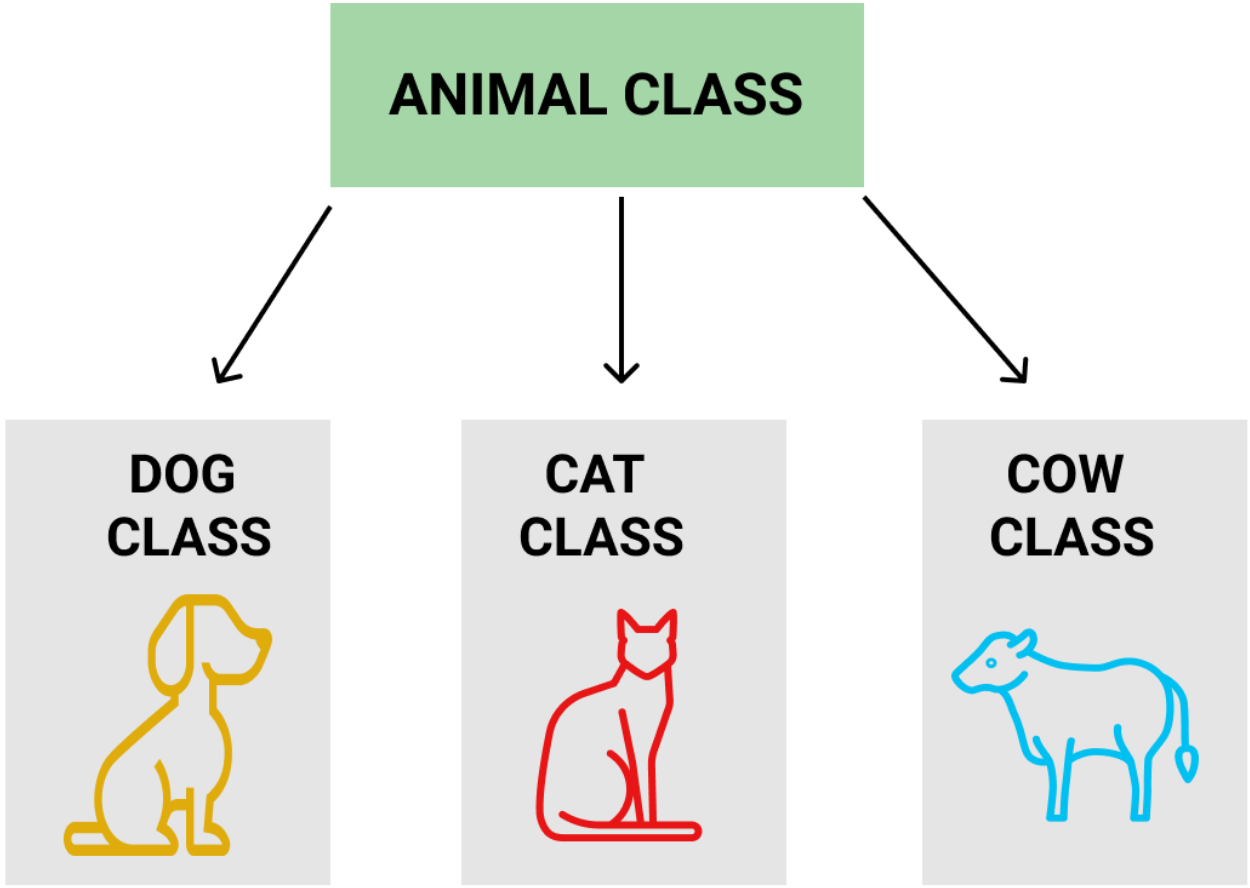
```
protected:
    int id_protected;
```

# 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.



# Implementing inheritance in C++:

- For creating a sub-class which is inherited from the base class we have to follow the below syntax.

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.

# Implementing inheritance in C++:

```
class Child : public Parent
{
    public:
     int id_c;
};
```

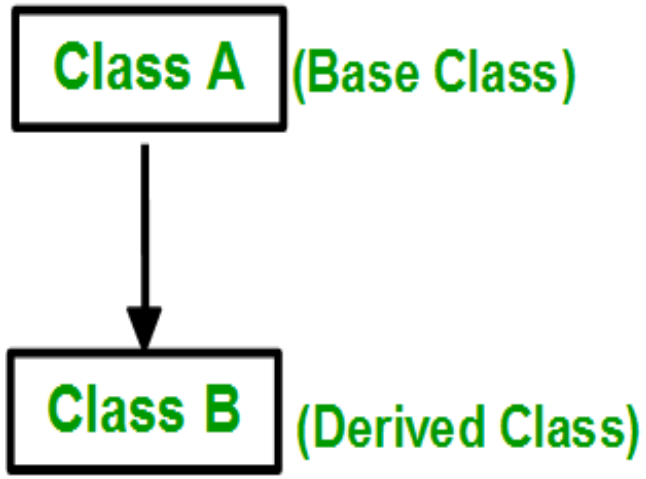
# Modes of Inheritance

- **Public mode:** If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
- **Protected mode:** If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
- **Private mode:** If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

# Modes of Inheritance

| Base class member access specifier | Type of Inheritence | | |
|---|---|---|---|
| | Public | Protected | Private |
| Public | Public | Protected | Private |
| Protected | Protected | Protected | Private |
| Private | Not accessible (Hidden) | Not accessible (Hidden) | Not accessible (Hidden) |

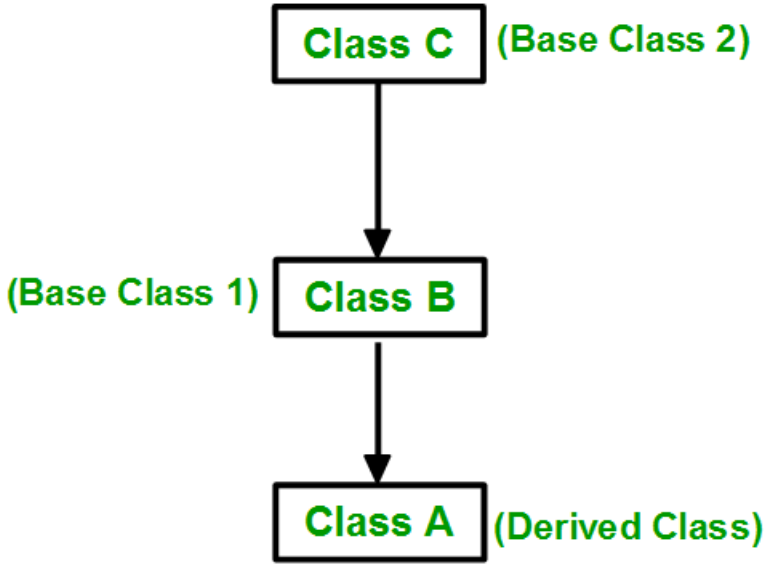
# Single Inheritance:




```
class subclass_name :
access_mode
base_class
{
  //body of subclass
};
```

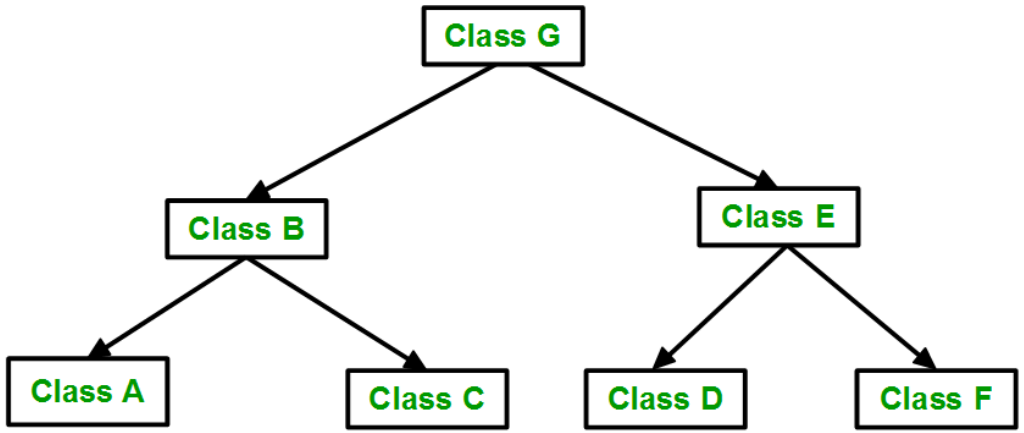
# Multilevel Inheritance:

- In this type of inheritance, a derived class is created from another derived class.



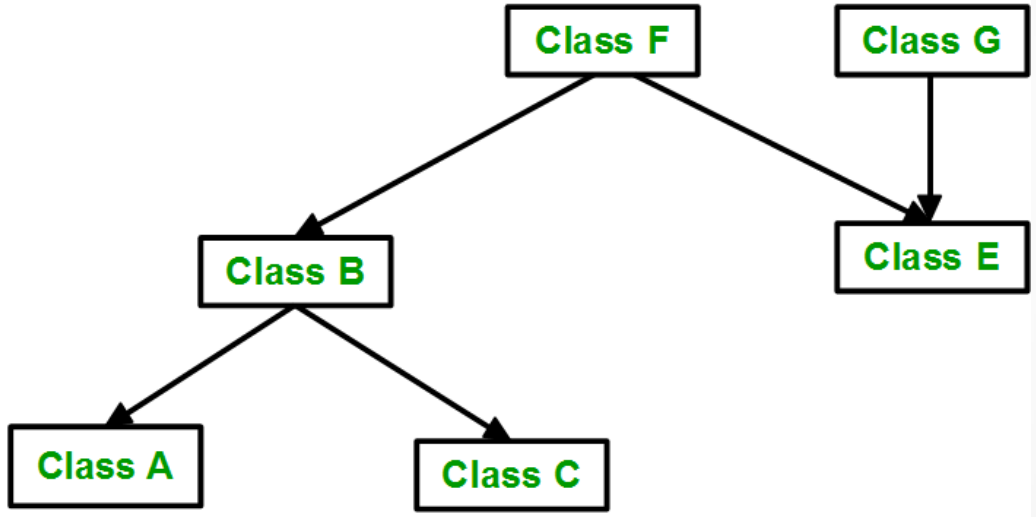
# Hierarchical Inheritance:

- In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



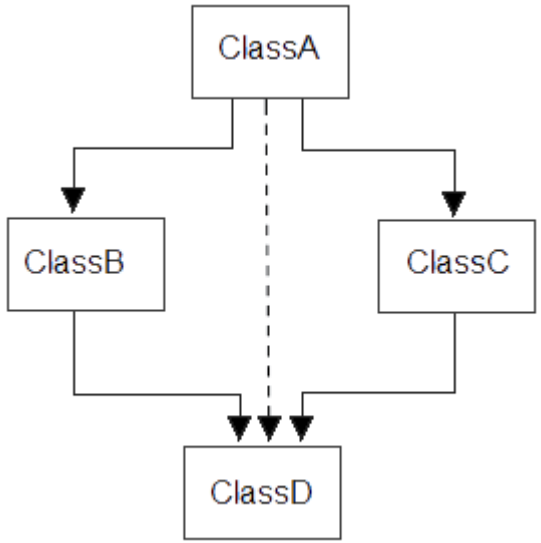
# Hybrid (Virtual) Inheritance:

- Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.
- Below image shows the combination of hierarchical and multiple inheritance:



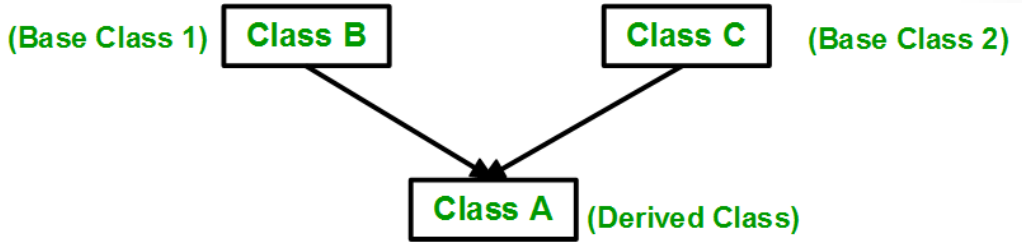
# A special case of hybrid inheritance : Multipath inheritance:

- A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. An ambiguity can arrise in this type of inheritance.



# Multiple Inheritance:

- Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.




```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  //body of subclass
};
```

# C++ supports operator overloading and function overloading.

- Operator Overloading: The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- Function Overloading: Function overloading is using a single function name to perform different types of tasks.
- Polymorphism is extensively used in implementing inheritance.

# Operator Overloading

- C++ also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator ‘+’ when placed between integer operands , adds them and when placed between string operands, concatenates them.

# Runtime polymorphism

This type of polymorphism is achieved by Function Overriding.

**Function Overriding-**on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

# Runtime polymorphism

- The main thing to note about the program is that the derived class's function is called using a base class pointer.
- The idea is that virtual functions are called according to the type of the object instance pointed to or referenced, not according to the type of the pointer or reference.
- In other words, virtual functions are resolved late, at runtime.

```
#include <iostream>
using namespace std;

class Base {
public:
    virtual void show()
    {
        cout << " In Base \n";
    }
};

class Derived : public Base {
public:
    void show()
    {
        cout << "In Derived \n";
    }
};
```

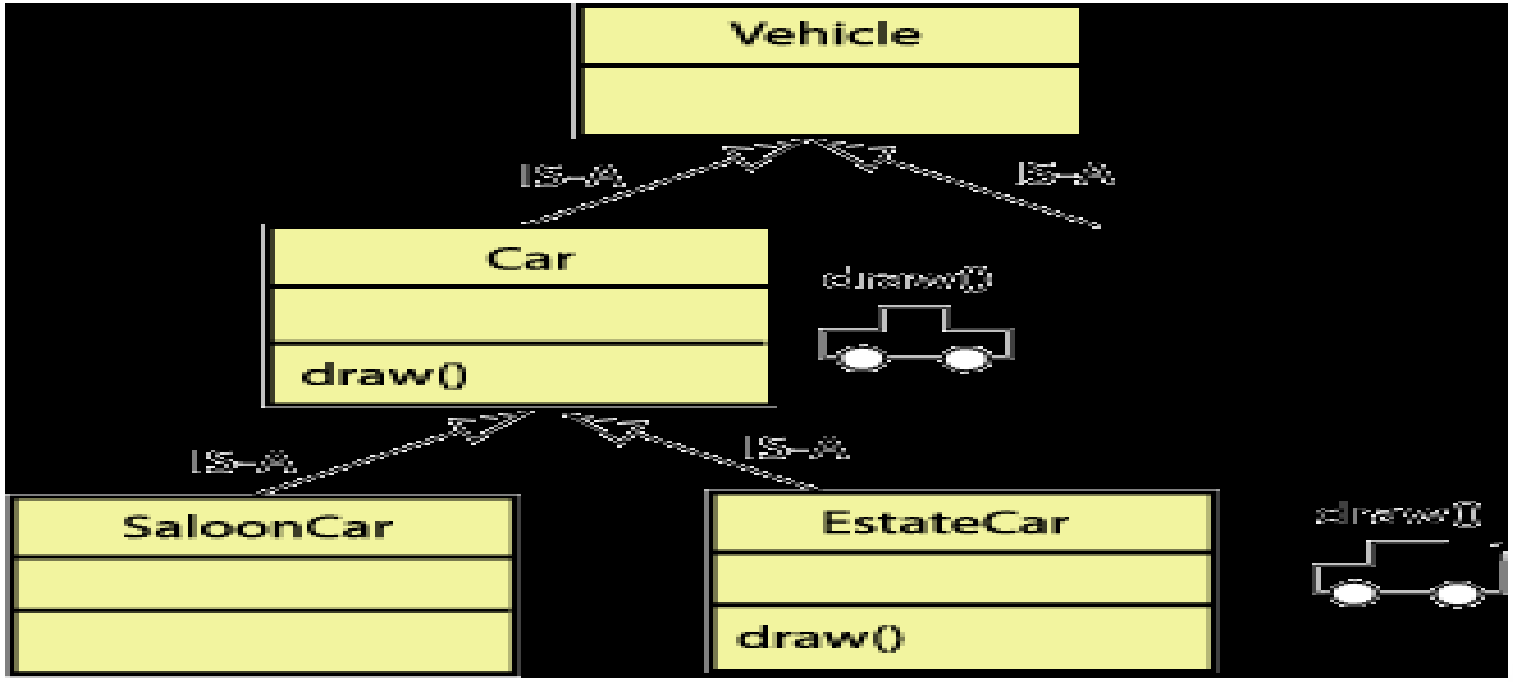
# Over-Riding

- As discussed, a derived class inherits its methods from the base class. It may be necessary to redefine an inherited method to provide specific behaviour for a derived class - and so alter the implementation. So, over-riding is the term used to describe the situation where the same method name is called on two different objects and each object responds differently.

- Over-riding allows different kinds of objects that share a common behaviour to be used in code that only requires that common behaviour.

# Over-Riding

- The over-ridden draw() method.



# Message Passing:

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

# UNIT 2

**C++ Tokens-**

A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

Keywords

Identifiers

Constants

Strings

Special Symbols

Operators

# Keyword:

Keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program.

C language supports 32 keywords which are given below:

auto double int struct
break else long switch
case enum register typedef
char extern return union
const float short unsigned
continue for signed void
default goto sizeof volatile
do if static while

# Keyword:

While in C++ there are 31 additional keywords other than C Keywords they are:

asm bool catch class
const_cast delete dynamic_cast explicit
export false friend inline
mutable namespace new operator
private protected public reinterpret_cast
static_cast template this throw
true try typeid typename
using virtual wchar_t

# Identifiers:

Identifiers are used as the general terminology for naming of variables, functions and arrays.

There are certain rules that should be followed while naming c identifiers:

- They must begin with a letter or underscore(_).
- They must consist of only letters, digits, or underscore. No other special character is allowed.
- It should not be a keyword.
- It must not contain white space.
- It should be up to 31 characters long as only first 31 characters are significant.

# Constants:

Constants are also like normal variables. But, only difference is, their values can not be modified by the program once they are defined.

Constants refer to fixed values. They are also called as literals.

Constants may belong to any of the data type.

**Syntax:**

const data_type variable_name; (or) const data_type *variable_name;

# Types of Constants:

- Integer constants – Example: 0, 1, 1218, 12482
- Real or Floating point constants – Example: 0.0, 1203.03, 30486.184
- Octal & Hexadecimal constants – Example: octal: (013 )8 = (11)10, Hexadecimal: (013)16 = (19)10
- Character constants -Example: ‘a’, ‘A’, ‘z’
- String constants -Example: “helloworld”

# Strings:

Strings are nothing but an array of characters ended with a null character (‘\0’).This null character indicates the end of the string. Strings are always enclosed in double quotes. Whereas, a character is enclosed in single quotes in C and C++.Declarations for String:

- char string[20] = {‘g’, ’e’, ‘e’, ‘k’, ‘s’, ‘f’, ‘o’, ‘r’, ‘g’, ’e’, ‘e’, ‘k’, ‘s’, ‘\0’};
- char string[20] = “geeksforgeeks”;
- char string [] = “geeksforgeeks”;

Difference between above declarations are:

- when we declare char as “string[20]”, 20 bytes of memory space is allocated for holding the string value.
- When we declare char as “string[]”, memory space will be allocated as per the requirement during execution of the program

# Special Symbols:

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.[] () {}, ; * = #

Brackets[]: Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.

Parentheses(): These special symbols are used to indicate function calls and function parameters.

Braces{}: These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.

comma (, ): It is used to separate more than one statements like for separating parameters in function calls.

semi colon : It is an operator that essentially invokes something called an initialization list.

asterisk (*): It is used to create pointer variable.

assignment operator: It is used to assign values.

pre processor(#): The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.

# Operators:

Operators are symbols that triggers an action when applied to C variables and other objects. The data items on which operators act upon are called operands.

Depending on the number of operands that an operator can act upon, operators can be classified as follows:

- Unary Operators: Those operators that require only single operand to act upon are known as unary operators.For Example increment and decrement operators

# Operators Classification

**Binary Operators:**

Those operators that require two operands to act upon are called binary operators. Binary operators are classified into :

Arithmetic operators

Relational Operators

Logical Operators

Assignment Operators

Conditional Operators

Bitwise Operators

Ternary Operators: These operators requires three operands to act upon. For Example Conditional operator(?:).

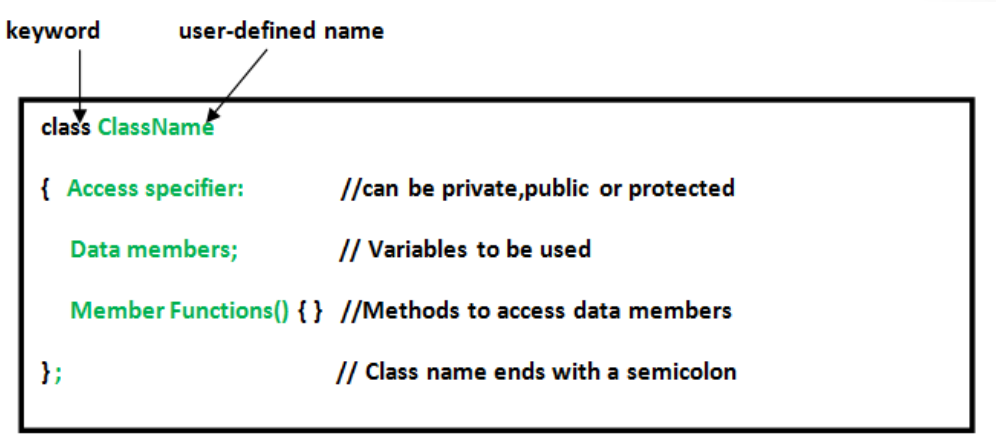
# C++ Classes and Objects

**Class:** A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

# Object

- An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.



# Declaring Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

ClassName ObjectName;

# Accessing data members and member functions:

The data members and member functions of class can be accessed using the dot(‘.’) operator with the object. For example if the name of object is obj and you want to access the member function with the name printName() then you will have to write obj.printName() .

**Accessing Data Members**

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.

This access control is given by Access modifiers in C++. There are three access modifiers : public, private and protected.

```
include <bits/stdc++.h>
using namespace std;
class Geeks
{
    // Access specifier
    public:

    // Data Members
    string geekname;

    // Member Functions()
    void printname()
    {
       cout << "Geekname is: " << geekname;
    }
};
```

```
int main() {

    // Declare an object of class geeks
    Geeks obj1;

    // accessing data member
    obj1.geekname = "main";

    // accessing member function
    obj1.printname();
    return 0;
}
```

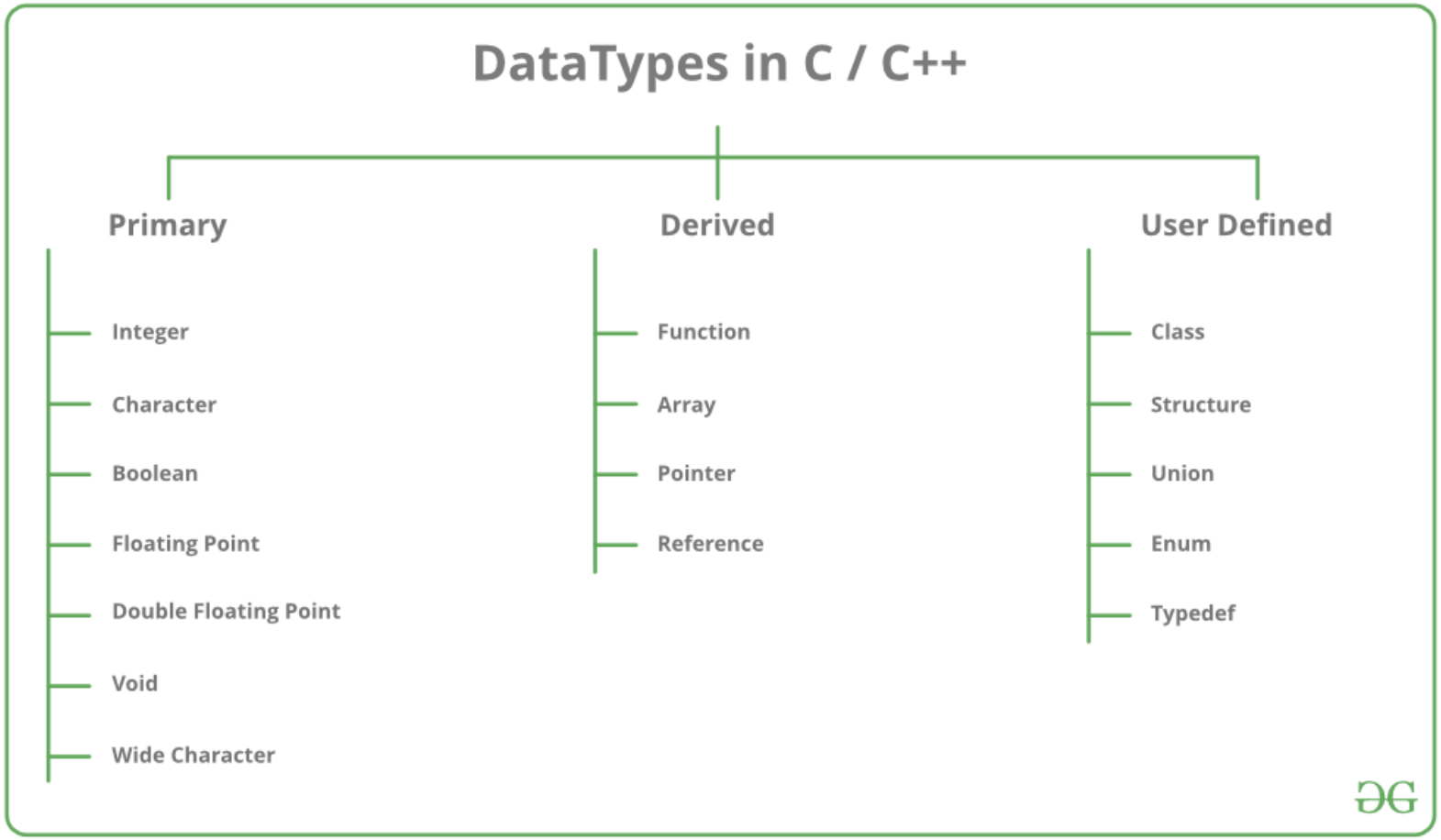
**Output:**

Geekname is: main

# C++ Data Types-

All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data-type with which it is declared. Every data type requires a different amount of memory.

# C++ Data Types-


DataTypes in C / C++
Primary
Integer
Character
Boolean
Floating Point
Double Floating Point
Void
Wide Character
Derived
Function
Array
Pointer
Reference
User Defined
Class
Structure
Union
Enum
Typedef

# Data types in C++ is mainly divided into three types:

**Primitive Data Types:** These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:

Integer

Character

Boolean

Floating Point

Double Floating Point

Valueless or Void

Wide Character

# DERIVED DATATYPES

The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:

Function

Array

Pointer

Reference

# USER DEFINED DATATYPE

These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:

Class

Structure

Union

Enumeration

Typedef defined DataType

# Data Abstraction & ADT(Abstract Datatypes)

- When you start your car, you don't need to know the intricate workings of the starter motor. All you need to do is turn the key to initiate the sequence. If successful, the engine will turn over. This real-world example highlights the programming concept of data abstraction, which allows a programmer to protect/hide the implementation of a process and only gives the keys to other functions or users. You only need to know enough about a given function to run it but don't need to know (or care) about how the internal code works.

- To take this a step further, we can create entire data types. An abstract data type (or ADT) is a class that has a defined set of operations and values. In other words, you can create the starter motor as an entire abstract data type, protecting all of the inner code from the user. When the user wants to start the car, they can just execute the start() function. In programming, an ADT has the following features:

# ADT(ABSTRACT DATATYPES)

- An ADT doesn't state how data is organized, and
- It provides only what's needed to execute its operations

An ADT is a prime example of how you can make full use of data abstraction and data hiding. This means that an abstract data type is a huge component of object-oriented programming methodologies: enforcing abstraction, allowing data hiding (protecting information from other parts of the program), and encapsulation (combining elements into a single unit, such as a class).

# Abstract Data Types

- Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.

- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

# Attributes in C++

Class attributes are just variables from a general programming point of view. But when it comes to Object Oriented Programming, these class attributes define the state of the class objects.

Following class defines a class named Student, with three attributes.

```
class Student {
   string name;
   int rollno;
   int section;
};
```

# C++ Class Methods

Methods of a class defines the behavior of the class objects. Class methods are functions that can be accessed within the class or on the class objects.

Following example, defines a class named Student with method printDetails().

```
class Student {
   //attributes
   string name;
   int rollno;
   int section;
  //methods
   void printDetails(){
      cout << "Name : " << name << endl;
      cout << "Roll Number : " << rollno << endl;
      cout << "Section : " << section<< endl;
   }
};
```

# Instantiate object

To instantiate is to create an instance of an object in an object-oriented programming (OOP) language. An instantiated object is given a name and created in memory or on disk using the structure described within a class declaration.

ex- A a1;

## Object Interaction

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

ex-a1.salary(name);

# Object lifetime

**Start-**If the object is of build int type and the definition has no initializer, no initialization takes place and start of the object's lifetime is the same as the start of its storage duration.

object lifetime starts immediately after the object has been initialized with that value, which means effectively immediately at start of the storage duration as well.

**End-**The end of an object's lifetime is determined exactly symmetrical to its start: If there is no destructor or if the destructor is trivial, the object's lifetime ends with its storage duration.if there is a destructor, the lifetime of the object stops as soon as the destructor body starts to execute.

# Constructors

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors initialize values to object members after storage is allocated to the object.

```
class A
{
    public:
    int x;
    // constructor
    A()
    {
        // object initialization
    }
};
```

While defining a contructor you must remeber that the name of constructor will be same as the name of the class, and contructors will never have a return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A
{
   public:
   int i;
   A(); // constructor declared
};

// constructor definition
A::A()
{
   i = 1;
}
```

# Types of Constructors in C++

Constructors are of three types:

- Default Constructor
- Parametrized Constructor
- Copy COnstructor

**Default Constructors-**Default constructor is the constructor which doesn't take any argument. It has no parameter.

**Syntax:**

```
class_name(parameter1, parameter2, ...)
{
    // constructor Definition
}
```

# For example:

```
class Cube
{
   public:
int side;
Cube()
 {

      side = 10;
   }
};

int main()
{
   Cube c;
   cout << c.side;}
```

**Output-10**

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

# Parameterized Constructors

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

For example:

```
Cube(int x)
   {
      side=x;
   }
```

```
class Cube
{
   public:
   int side;
   Cube(int x)
   {
      side=x;
   }
};

int main()
{
   Cube c1(10);
   Cube c2(20);
   Cube c3(30);
   cout << c1.side;
   cout << c2.side;
   cout << c3.side;
}
```

**output**

10
20
30

By using parameterized construcor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

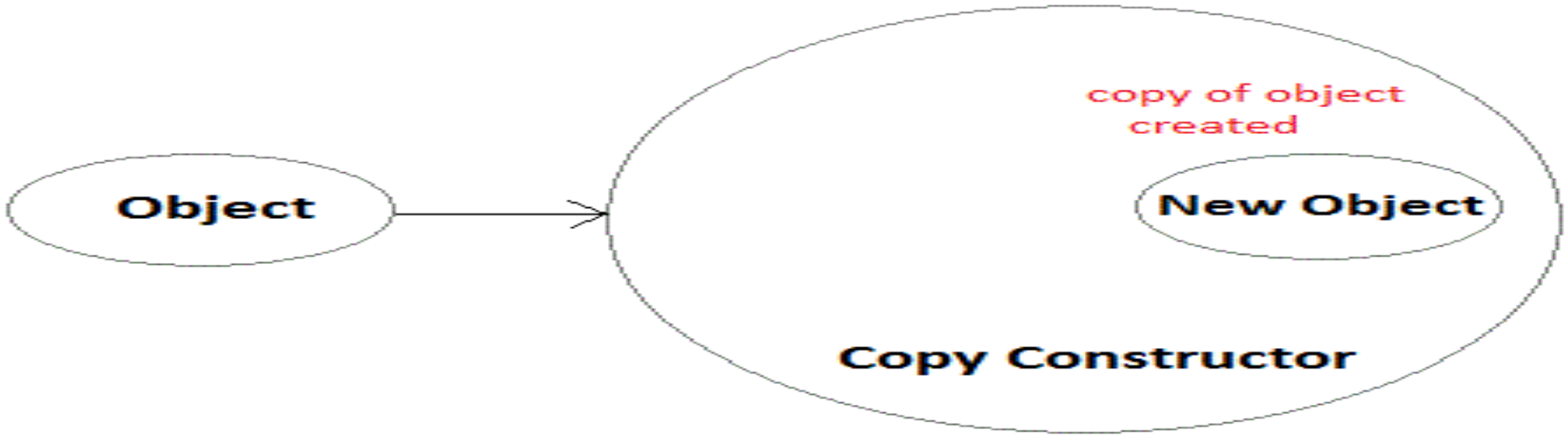
# Copy Constructors

Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form X (X&), where X is the class name. The compiler provides a default Copy Constructor to all the classes.

**Syntax**

```
Classname(const classname & objectname)
{
    . . . .
}
```

As it is used to create an object, hence it is called a constructor. And, it creates a new object, which is exact copy of the existing copy, hence it is called copy constructor.



```
#include<iostream>
using namespace std;
class Samplecopyconstructor
{
   private:
   int x, y;   //data members

   public:
   Samplecopyconstructor(int x1, int y1)
   {
      x = x1;
      y = y1;
   }
```

```
/* Copy constructor */
   Samplecopyconstructor (const Samplecopyconstructor &sam)
   {
      x = sam.x;
      y = sam.y;
   }

   void display()
   {
      cout<<x<<" "<<y<<endl;
   }
};
/* main function */
int main()
{
   Samplecopyconstructor obj1(10, 15);     // Normal constructor
   Samplecopyconstructor obj2 = obj1;      // Copy constructor
   cout<<"Normal constructor : ";
   obj1.display();
   cout<<"Copy constructor : ";
   obj2.display();
   return 0;
}
```

**OUTPUT**

Normal constructor :10 15

Copy constructor : 10 15

# Destructors in C++

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a tilde ~ sign as prefix to it.

```
class A
{
   public:
   // defining
destructor for
class
   ~A()
   {
      // statement
   }
};
```

Destructors will never have any arguments.

# Example to see how Constructor and Destructor are called

```
class A
{
   // constructor
   A()
   {
      cout << "Constructor called";
   }

   // destructor
   ~A()
   {
      cout << "Destructor called";
   }
};
```

```
int main()
{
   A obj1;   // Constructor
Called
   int x = 1
   if(x)
   {
      A obj2;  //
Constructor Called
   }   // Destructor Called
for obj2
} //  Destructor called for
obj1
```

**Output**-

Constructor called
Constructor called
Destructor called
Destructor called

# Constructor and Destructor

When an object is created the constructor of that class is called. The object reference is destroyed when its scope ends, which is generally after the closing curly bracket } for the code block in which it is created.

When an object is created the constructor of that class is called. The object reference is destroyed when its scope ends, which is generally after the closing curly bracket } for the code block in which it is created.

# static object and dynamic object

```
class cat
{
  private:
     int age;
  public:
     cat();
};
int main(void)
{
  cat object; // static object
  cat *pointer = new cat(); // dynamic object
}
```

In C++, **dynamic memory allocation** is done by using the new and delete operators. There is a similar feature in C using malloc(), calloc(), and deallocation using the free() functions. Note that these are functions. This means that they are supported by an external library. C++, however, imbibed the idea of dynamic memory allocation into the core of the language feature by using the new and delete operators. Unlike C's dynamic memory allocation and deallocation functions, new and delete in C++ are operators and are a part of the list of keywords used in C++.

# Dynamic Memory Management

- The allocation or deallocation of an object—be it arrays of any built-in type or any user-defined type—can be controlled in memory at runtime by using the new and delete operators.

**The new Operator**

The new operator allocates the exact amount of memory required by the object during execution. The allocation is performed on the free memory area, called the heap, assigned to each and every program especially for the purpose of dynamic memory allocation. The allocation, if successful by using the new operator, returns a pointer to the object; if it fails, a bad_alloc exception is raised. However, on success we can access the object via pointer returned by the new operator.

**IntArray *intArray = new IntArray(10);**

**MyClass *obj = MyClass;**

# The new Operator success

- What happens here is that the new operator allocates the exact amount of memory required by the object in the free store or heap area, invokes the default constructor to initialize the properties of the object, and returns a pointer to the type-specified object allocated in the memory. However, during the allocation process, if the operator finds that there is not enough space in memory to allocate for the object, it throws an exception to indicate the error. The exception gives an opportunity to handle the error programmatically.

# The delete Operator

- As soon as we do not need the allocated space, such as, due to object going out of scope, and so forth, we can ensure that the memory occupied is returned back to the store or freed by using the delete operator. The freed memory then again can be reused by succeeding new operator calls. For example, if we write as follows,

**double *dArray = new double[50]{};**

**MyClass *obj = new MyClass;**

The empty braces indicate that each element must be initialized with its default initialization. The default initialization of fundamental types is 0.Now, to deallocate the memory we use the following statement:

delete [] dArray;

delete obj;

# Scope of Variables in C++

There are mainly two types of variable scopes:

Local Variables

Global Variables

**Local Variables-**Variables defined within a function or block are said to be local to those functions.

Anything between '{' and '}' is said to inside a block.

Local variables do not exist outside the block in which they are declared, i.e. they can not be accessed or used outside that block.

Declaring local variables: Local variables are declared inside a block.

```
#include<iostream>
using namespace std;
```
Global Variable
```
// global variable
int global = 5;

// main function
int main()
```
Local variable
```
{
    // local variable with same
    // name as that of global variable
    int global = 2;

    cout << global << endl;
}
```

# Scope of Variables in C++

**Global Variables-**As the name suggests, Global Variables can be accessed from any part of the program.They are available through out the life time of a program.

They are declared at the top of the program outside all of the functions or blocks.

Declaring global variables: Global variables are usually declared outside of all of the functions and blocks, at the top of the program. They can be accessed from any portion of the program.

filter_none

```
#include<iostream>
using namespace std;
 int global = 5;
 void display()
{
   cout<<global<<endl;
}
 int main()
{
   display();
    global = 10;
   display();
}
```

**Output**-5 10

# What if there exists a local variable with the same name as that of global variable inside a function?

```
#include<iostream>
using namespace std;
  int global = 5;
  int main()
{
      int global = 2;
    cout << global << endl;
}
```

- global variable then the compiler will give precedence to the local variable
- Here in the above program also, the local variable named “global” is given precedence. So the output is 2.

## Local vs Global Object in C++

```
#include <iostream>
 using namespace std;
class zzz {
   public:
      void show(int x) {
         cout << " x= " << x;
      }
};
zzz a; //  creating global object “a”.

int main() {
   zzz O; // creating local object “O”
   a.show(10); // call using global object
   O.show(5); // call using local object
}
```

**Output**-x=10  x=5

# C++ Array of Objects

In the following example, we shall declare an array of type Student, with size of five. Student is a class that we defined in the program. Then we shall assign objects using index of array.

```
class Student {
  public:
   string name;
   int rollno;
 Student() {}
 Student(string x, int y) {
     name = x;
      rollno = y;}
 void printDetails() {
cout << rollno << " - " << name << endl;
   }
};
```

# Array of Objects

```
int main() {
   //declare array with specific size
Student students[5];
  //assign objects
    students[0] = Student("Ram", 5);
    students[1] = Student("Alex", 1);
    students[2] = Student("Lesha", 4);
    students[3] = Student("Emily", 3);
    students[4] = Student("Anita", 2);
  for(int i=0; i < 5; i++) {
      students[i].printDetails();
    }
}
```

# inline function

C++ inline function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

function definition in a class definition is an inline function definition, even without the use of the inline specifier.

Following is an example, which makes use of inline function to return max of two numbers −

# inline function

```
#include <iostream>
 using namespace std;
inline int Max(int x, int y) {
   return (x > y)? x : y;
}
// Main function for the program
int main() {
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
    return 0;
}
```

# inline function

- When the above code is compiled and executed, it produces the following result –

Max (20,10): 20

Max (0,200): 200

Max (100,1010): 1010

# Static Keyword

- Static is a keyword in C++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime. Static Keyword can be used with following,
- Static variable in functions
- Static Class Objects
- Static member Variable in class
- Static Methods in class

# Static Variables inside Functions

Static variables when used inside function are initialized only once, and then they hold there value even through function calls.

These static variables are stored on static storage area , not in stack.

```
void counter(){
    static int count=0;
 cout << count++;
}
int main()
{ for(int i=0;i<5;i++)
    {counter(); }
}
```

**Output-0 1 2 3 4**

# Let's se the same program's output without using static variable.

```
void counter()
{
   int count=0;
   cout << count++;
}

int main(0
{
   for(int i=0;i<5;i++)
   {
      counter();
   }
}
```

**OUTPUT-  0 0 0 0 0**

# Static Variables inside Functions

- If we do not use static keyword, the variable count, is reinitialized everytime when counter() function is called, and gets destroyed each time when counter() functions ends. But, if we make it static, once initialized count will have a scope till the end of main() function and it will carry its value through function calls too.
- If you don't initialize a static variable, they are by default initialized to zero.

# Static Class Objects

- Static keyword works in the same way for class objects too. Objects declared static are allocated storage in static storage area, and have scope till the end of program.
- Static objects are also initialized using constructors like other normal objects. Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

# Static Class Objects

```
class Abc
{int i;
   public:
    Abc()
    {
       i=0;
       cout << "constructor";
    }
    ~Abc()
    {
       cout << "destructor";
    }
};
void f()
{
    static Abc obj;
}
```

# Static Class Objects

```
int main()
{
   int x=0;
   if(x==0)
   {
      f();
   }
   cout << "END";
}
```

**OUTPUT-  constructor END destructor**

why was the destructor not called upon the end of the scope of if condition, where the reference of object obj should get destroyed. This is because object was static, which has scope till the program's lifetime, hence destructor for this object was called when main() function exits.

# Static Data Member in Class

- Static data members of class are those members which are shared by all the objects. Static data member has a single piece of storage, and is not available as separate copy with each object, like other non-static data members.

- Static member variables (data members) are not initialied using constructor, because these are not dependent on object initialization.

- Also, it must be initialized explicitly, always outside the class. If not initialized, Linker will give error.

# Static Data Member in Class

```
class X
{public:
    static int i;
    X()
    {
        // construtor
    }};
int X::i=1;
int main()
{
    X obj;
    cout << obj.i;   // prints value of i
}
```

**OUTPUT-1**

Once the definition for static data member is made, user cannot redefine it.

# Static Member Functions

- These functions work for the class as whole rather than for a particular object of a class.

- It can be called using an object and the direct member access . operator. But, its more typical to call a static member function by itself, using class name and scope resolution :: operator.

```
class X
{
   public:
   static void f()
   {
      // statement
   }
};

int main()
{
   X::f();   // calling member function directly with class name
}
```

These functions cannot access ordinary data members and member functions, but only static data members and static member functions.

# friend function

- A friend function can access the private and protected data of a class. We declare a friend function using the friend keyword inside the body of the class.

```
class className {
    … .. …
    friend returnType functionName(arguments);
    … .. …
}
```

# Friend class and function

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

```
class Node {
private:
   int key;
   Node* next;
   /* Other members of Node Class */

   // Now class  LinkedList can
   // access private members of Node
   friend class LinkedList;
};
```

# Friend Function

Friend Function Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:
a) A method of another class
b) A global function

```
class Node
 {
private:
   int key;
   Node* next;
 /* Other members of Node Class */
   friend int LinkedList::search();
   // Only search() of linkedList
   // can access internal members
};
```

# Following are some important points about friend functions and classes:

1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

2) Friendship is not mutual. If class A is a friend of B, then B doesn’t become a friend of A automatically.

3) Friendship is not inherited

4) The concept of friends is not there in Java.

# A simple and complete C++ program to demonstrate friend Class

```
#include <iostream>
class A {
private:
    int a;
public:
    A() { a = 0; }
    friend class B; // Friend Class
};
class B {
private:
    int b;
public:
   void showA(A& x)
    {
      // Since B is friend of A, it can access
        // private members of A
        std::cout << "A::a=" << x.a;
    }
};
```

# A simple and complete C++ program to demonstrate friend Class

```
int main()
{
    A a;
    B b;
    b.showA(a);
    return 0;
}
```

Output:  A::a=0

# A simple and complete C++ program to demonstrate friend function of another class

```
class B;
  class A {
public:
 void showB(B&);
};
  class B {
private:
    int b;
  public:
    B() { b = 0; }
    friend void A::showB(B& x); // Friend function
};
```

# A simple and complete C++ program to demonstrate friend function of another class

```
void A::showB(B& x)
{
    // Since showB() is friend of B, it can
    // access private members of B
    std::cout << "B::b = " << x.b;
}
  int main()
{
   A a;
    B x;
    a.showB(x);
    return 0;
}
```

Output:  B::b = 0

# A simple and complete C++ program to demonstrate global friend

```
class A {
    int a;
  public:
    A() { a = 0; }
  // global friend function
    friend void showA(A&);
};
  void showA(A& x)
{ // Since showA() is a friend, it can access
    // private members of A
    std::cout << "A::a=" << x.a; }
```

# A simple and complete C++ program to demonstrate global friend

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

Output:

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

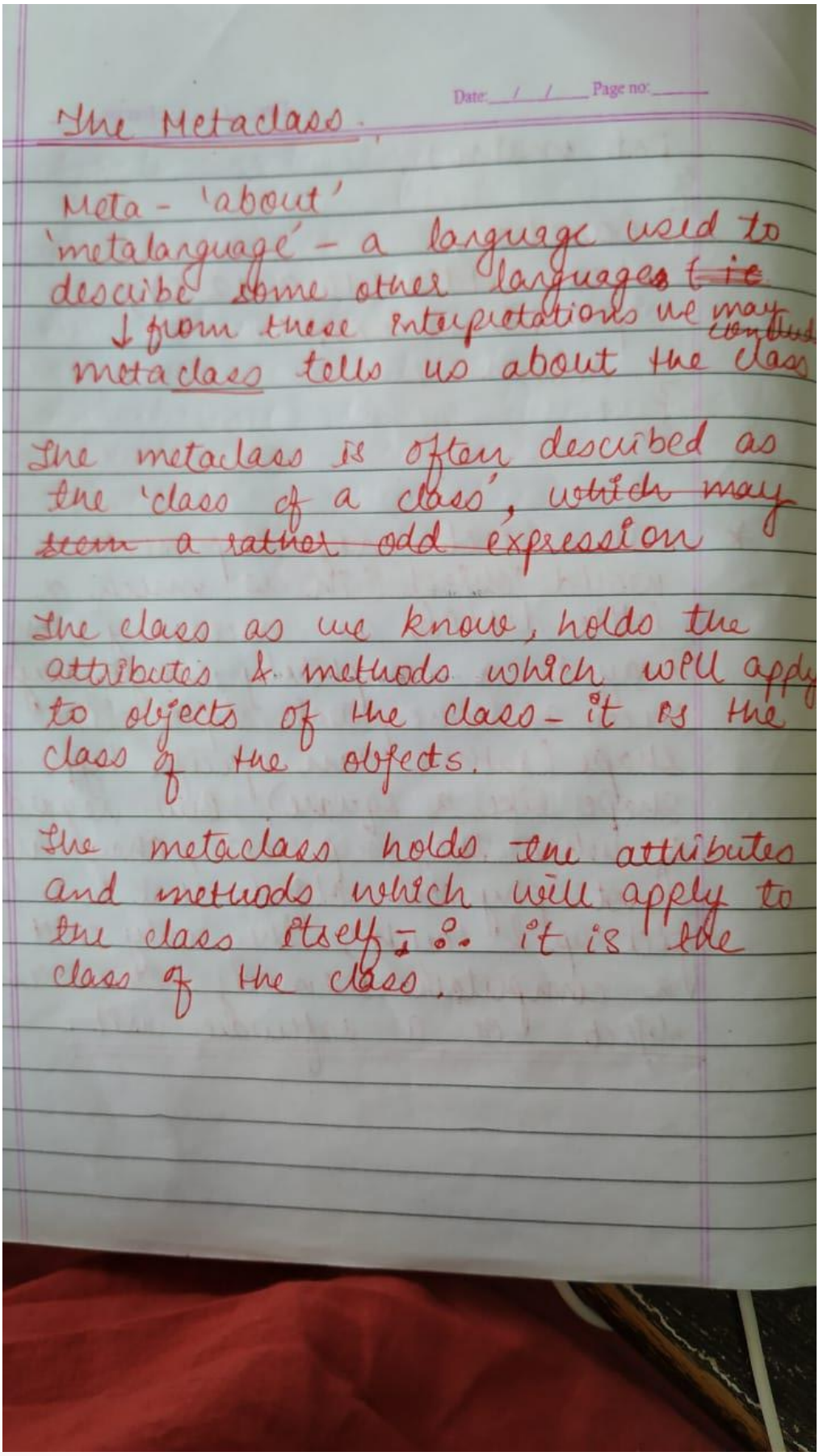
# Meta-Class

A Meta-Class is a class' class. If a class is an object, then that object must have a class (in classical OO anyway). Compilers provide an easy way to picture Meta-Classes. Classes must be implemented in some way; perhaps with dictionaries for methods, instances, and parents and methods to perform all the work of being a class. This can be declared in a class named "Meta-Class". The Meta-Class can also provide services to application programs, such as returning a set of all methods, instances or parents for review (or even modification). [Booch 91, p 119] provides another example in Smalltalk with timers. In Smalltalk, the situation is more complex. To make this easy, refer to the following listing, which is based on the number of levels of distinct instantiations:

# Meta-Class class

The "class class"es handle messages to classes, such as constructors and "new", and also "class variables" (a term from Smalltalk), which are variables shared between all instances of a class (static member data in C++). There are 3 distinct kinds of objects (objects, classes, and metaclasses).

# Meta Class



The Metaclass.

Date: / / Page no:

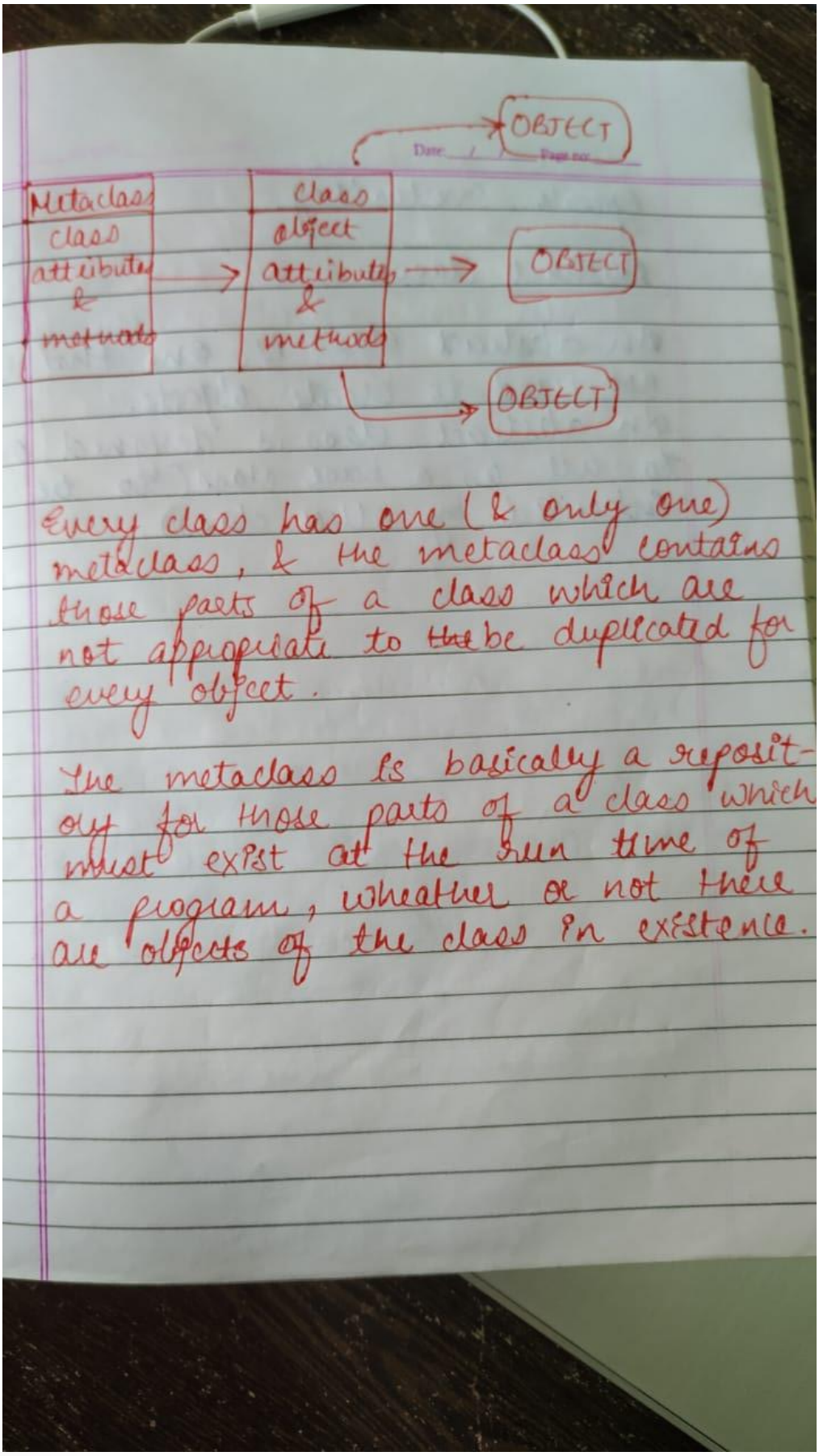
Meta - 'about'
'metalanguage' - a language used to describe some other languages ~~i.e.~~
↓ from these interpretations we may conclude
metaclass tells us about the class

The metaclass is often described as the 'class of a class', which may ~~seem a rather odd expression~~

The class as we know, holds the attributes & methods which will apply to objects of the class - it is the class of the objects.

The metaclass holds the attributes and methods which will apply to the class itself ∴ it is the class of the class.

# Meta Class




Every class has one (& only one) metaclass, & the metaclass contains those parts of a class which are not appropriate to ~~the~~ be duplicated for every object.

The metaclass is basically a repository for those parts of a class which must exist at the run time of a program, wheather or not there are objects of the class in existence.

# Unit 3

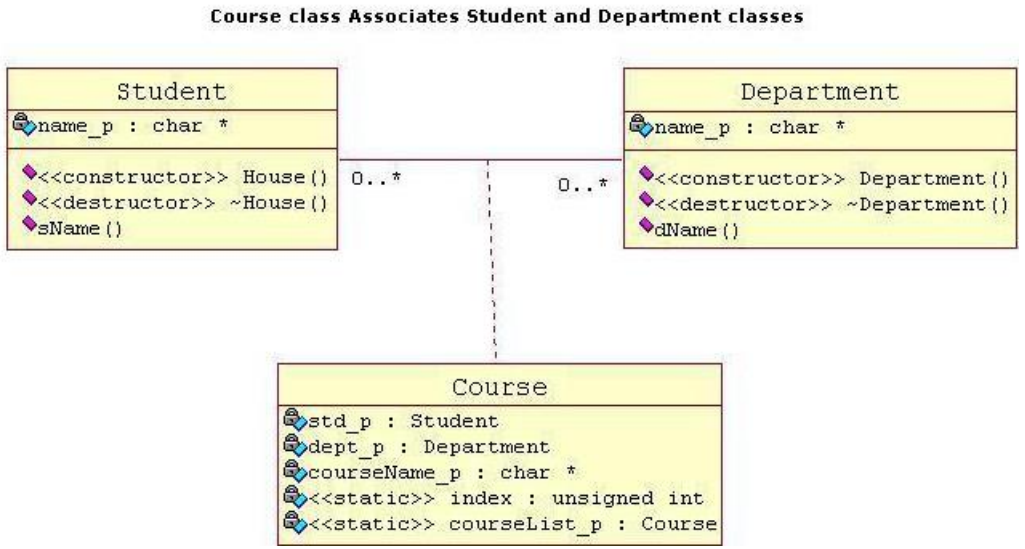
**Association** is a simple structural connection or channel between classes and is a relationship where all objects have their own lifecycle and there is no owner.

Lets take an example of Department and Student.

Multiple students can associate with a single Department and single student can associate with multiple Departments, but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.

Here is respective Model and Code for the above example.

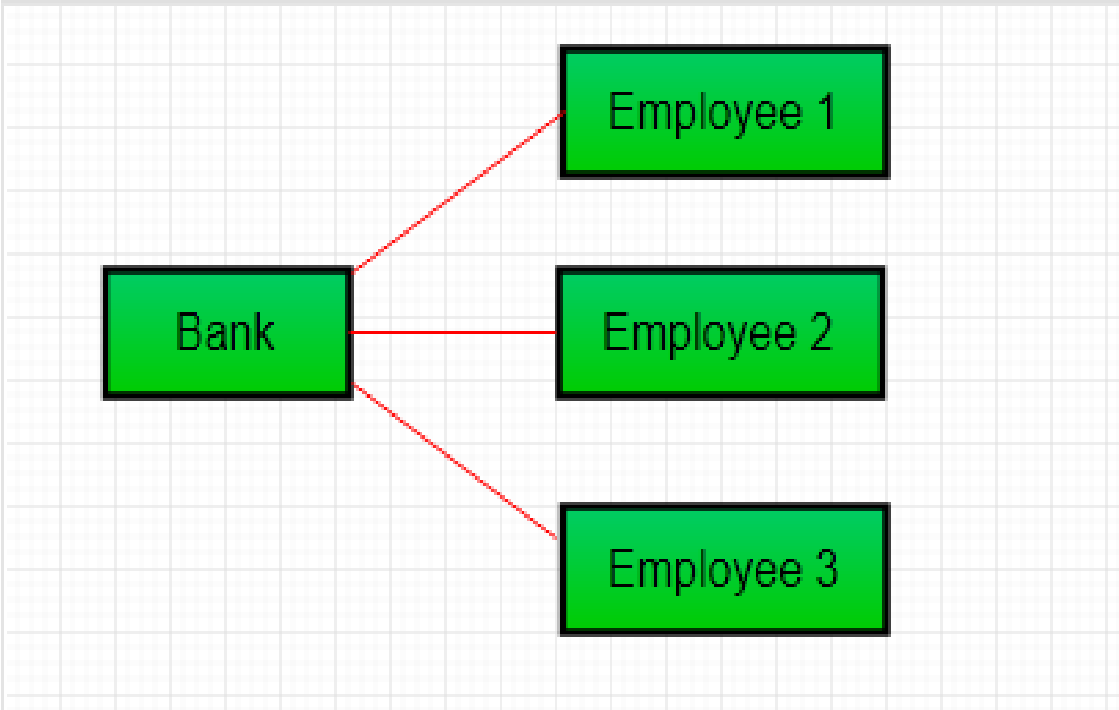
# Association



# Association

Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. Composition and Aggregation are the two forms of association.

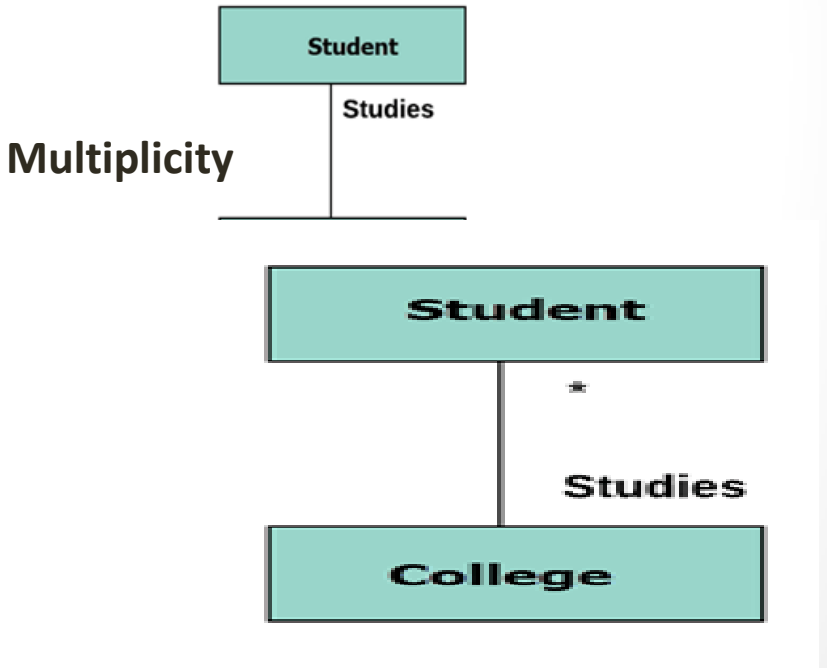
# Association




In above example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.

# Association:

- This kind of relationship represents static relationships between classes A and B. For example; an employee works for an organization.
- Association is mostly verb or a verb phrase or noun or noun phrase.
- It should be named to indicate the role played by the class attached at the end of the association path.



# Aggregation

Aggregation is a specialize form of Association where all object have their own lifecycle but there is a ownership like parent and child. Child object can not belong to another parent object at the same time. We can think of it as "has-a" relationship.

Implementation details:

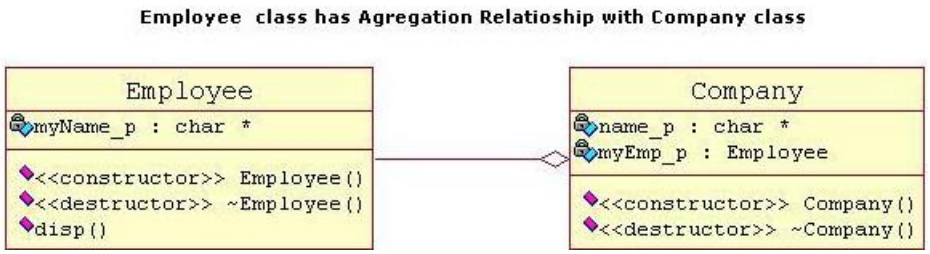
Typically we use pointer variables that point to an object that lives outside the scope of the aggregate class Can use reference values that point to an object that lives outside the scope of the aggregate class Not responsible for creating/destroying subclasses

Lets take an example of Employee and Company.

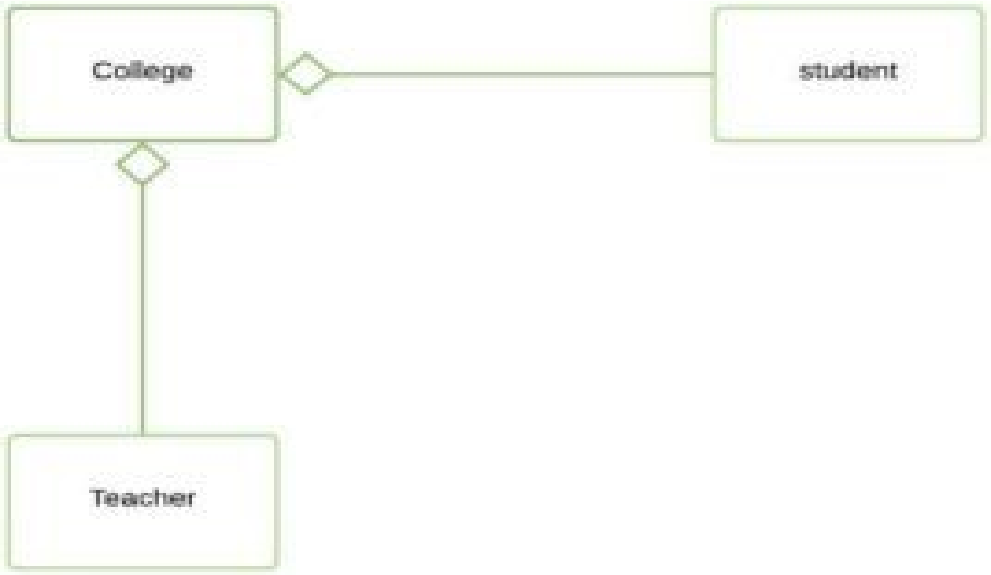
A single Employee can not belong to multiple Companies (legally!! ), but if we delete the Company, Employee object will not destroy.

Here is respective Model and Code for the above example.

# Aggregation



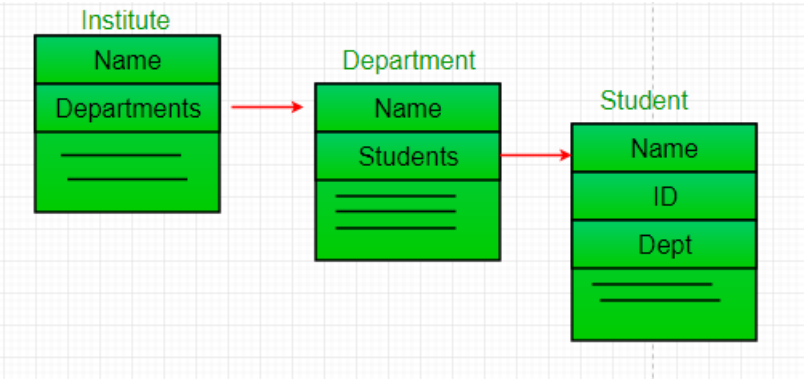
# Aggregation




It is a special form of Association where:
It represents Has-A relationship.
It is a unidirectional association i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
In Aggregation, both the entries can survive individually which means ending one entity will not effect the other entity

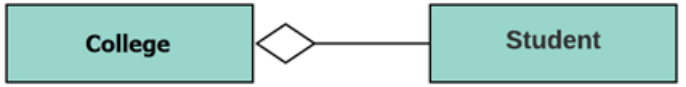
# Aggregation

- In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make a Institute class which has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of Student class means it is associated with Student class through its Object(s).
- It represents a Has-A relationship.



# Aggregation

- Aggregation is a special type of association that models a whole- part relationship between aggregate and its parts.
- For example, the class college is made up of one or more student. In aggregation, the contained classes are never totally dependent on the lifecycle of the container. Here, the college class will remain even if the student is not available.



# Composition

Composition is again specialize form ofAggregation. It is a strong type of Aggregation. Here the Parent and Child objects have coincident lifetimes. Child object dose not have it's own lifecycle and if parent object gets deleted, then all of it's child objects will also be deleted.

Implentation details:

1. Typically we use normal member variables

2. Can use pointer values if the composition class automatically handles allocation/deallocation

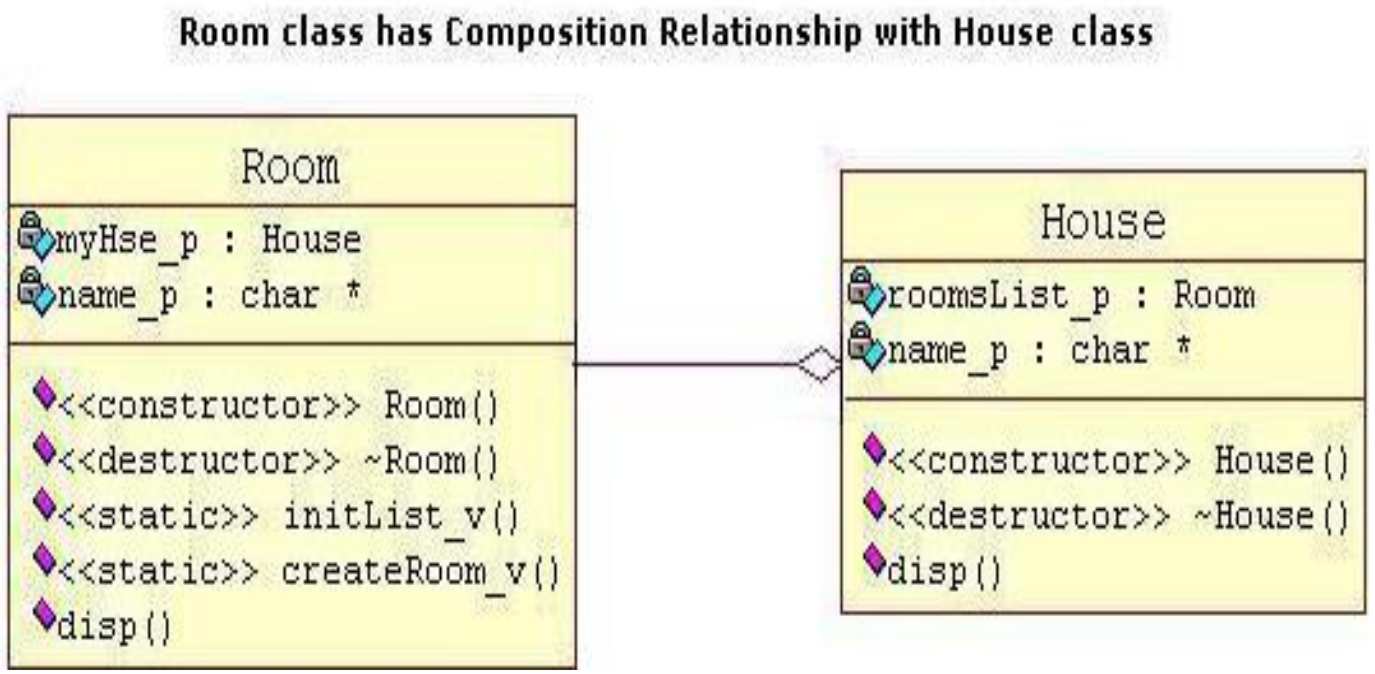
3. Responsible for creation/destruction of subclasses

Lets take an example of a relationship between House and it's Rooms.

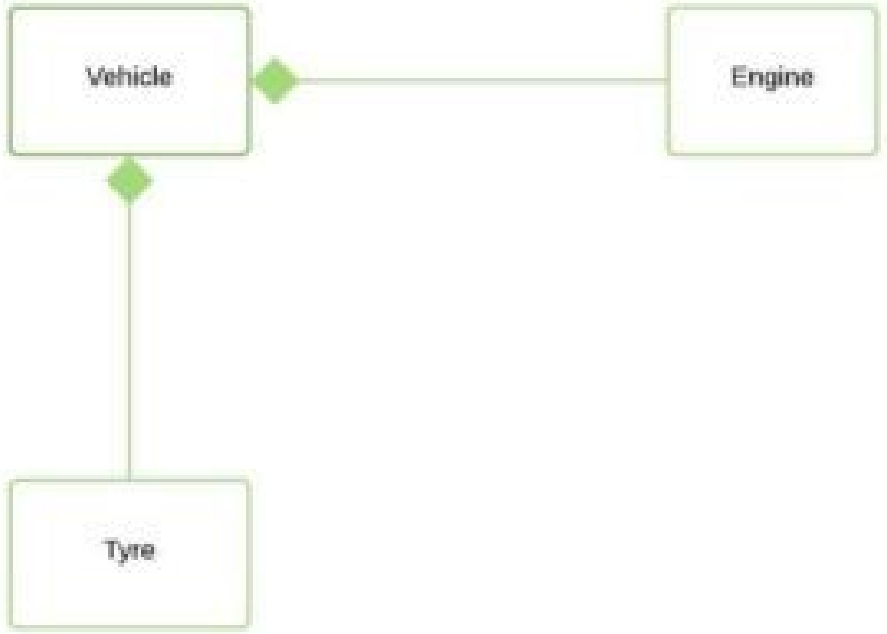
House can contain multiple rooms there is no independent life for room and any room can not belong to two different house. If we delete the house room will also be automatically deleted.

Here is respective Model and Code for the above example.

# Composition



# Composition




Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.It represents part-of relationship.In composition, both the entities are dependent on each other.

When there is a composition between two entities, the composed object cannot exist without the other entity.

Lets take example of Library.

# Composition

- a library can have no. of books on same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. book can not exist without library. That’s why it is composition.

# Composition:

- The composition is a special type of aggregation which denotes strong ownership between two classes when one class is a part of another class.


- For example, if college is composed of classes student. The college could contain many students, while each student belongs to only one college. So, if college is not functioning all the students also removed.

# Aggregation vs. Composition

**Aggregation**

Aggregation indicates a relationship where the child can exist separately from their parent class. Example: Automobile (Parent) and Car (Child). So, If you delete the Automobile, the child Car still exist.

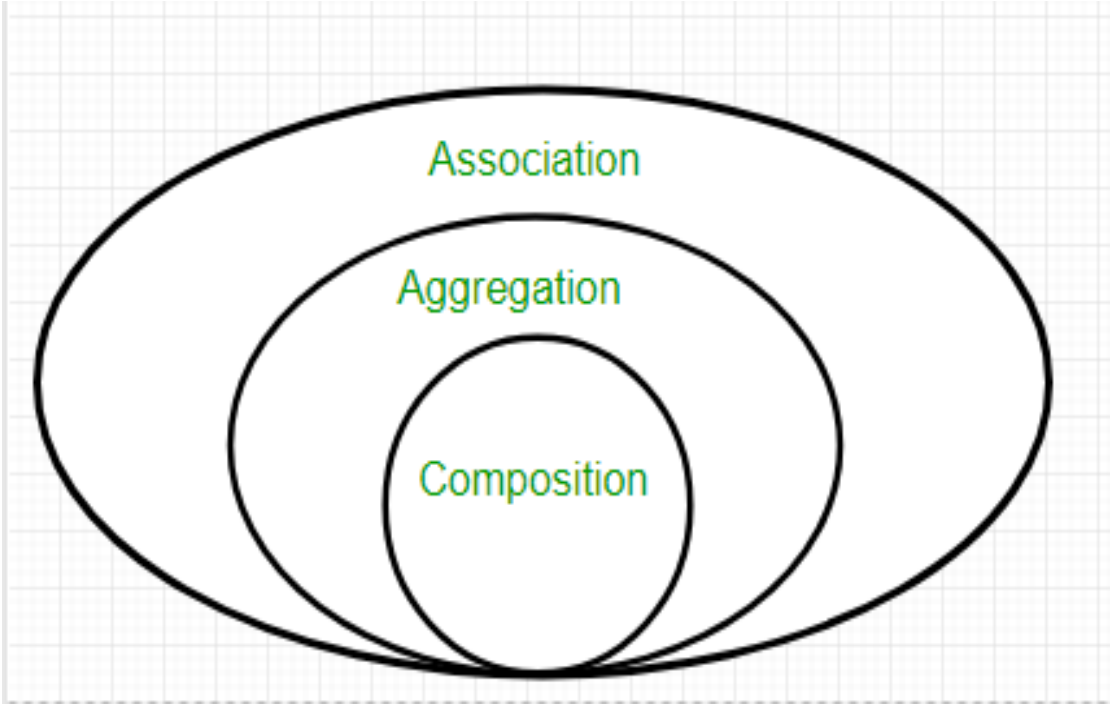
**Composition**

Composition display relationship where the child will never exist independent of the parent. Example: House (parent) and Room (child). Rooms will never separate into a House.

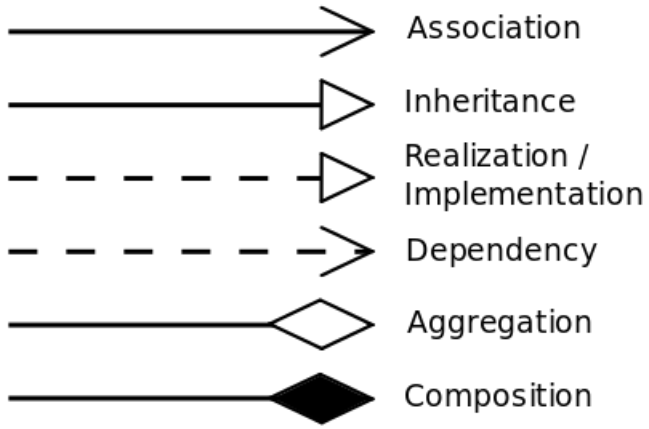
# Aggregation vs Composition

- **Dependency:** Aggregation implies a relationship where the child can exist independently of the parent. For example, **Bank and Employee,** delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: **Human and heart**, heart don't exist separate to a Human
- **Type of Relationship:** Aggregation relation is "has-a" and composition is "part-of" relation.
- **Type of association:** Composition is a strong Association whereas Aggregation is a weak Association.

# Association, Composition and Aggregation

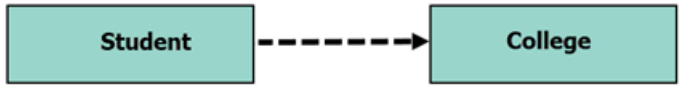


# Relationships



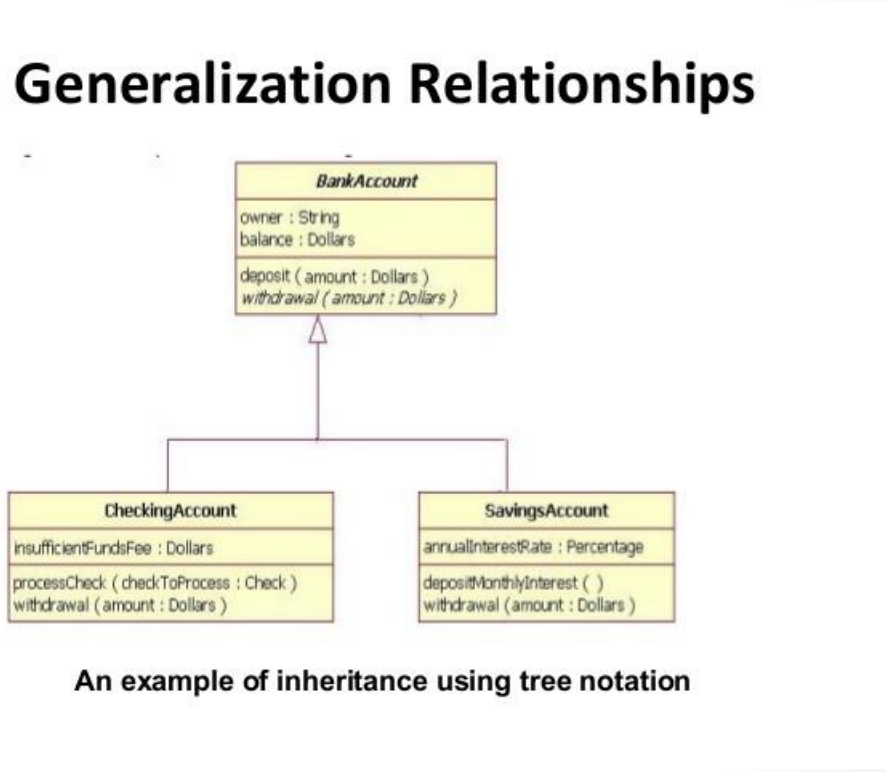
# Dependency

A dependency means the relation between two or more classes in which a change in one may force changes in the other. However, it will always create a weaker relationship. Dependency indicates that one class depends on another.



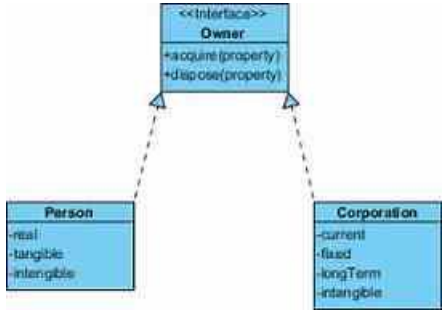
# Generalization(Inheritance)

- It is also called a parent-child relationship. In generalization, one element is a specialization of another general component. It may be substituted for it. It is mostly used to represent inheritance.

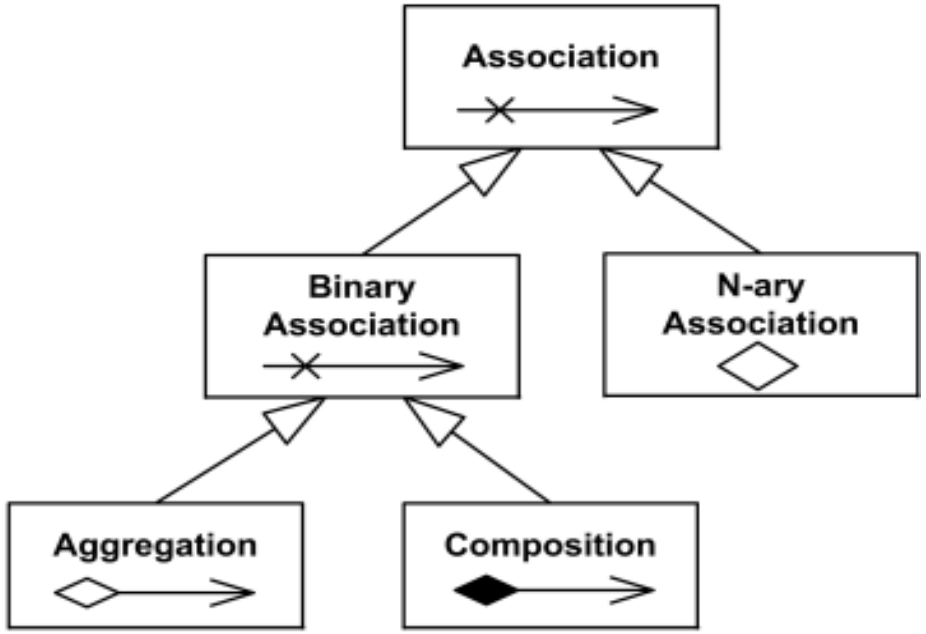


# Realization

In a realization relationship of UML, one entity denotes some responsibility which is not implemented by itself and the other entity that implements them. This relationship is mostly found in the case of interfaces.



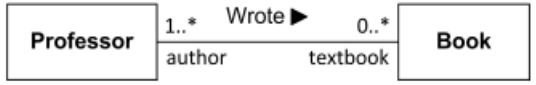
# Association relationship overview diagram




An association is usually drawn as a solid line connecting two classifiers or a single classifier to itself. Name of the association can be shown somewhere near the middle of the association line but not too close to any of the ends of the line. Each end of the line could be decorated with the name of the association end

# Association End

Association end is a connection between the line depicting an association and the icon depicting the connected classifier. Name of the association end may be placed near the end of the line. The association end name is commonly referred to as role name (but it is not defined as such in the UML 2.4 standard). The role name is optional and




Professor "playing the role" of author is associated with textbook end typed as Book.

# Association End

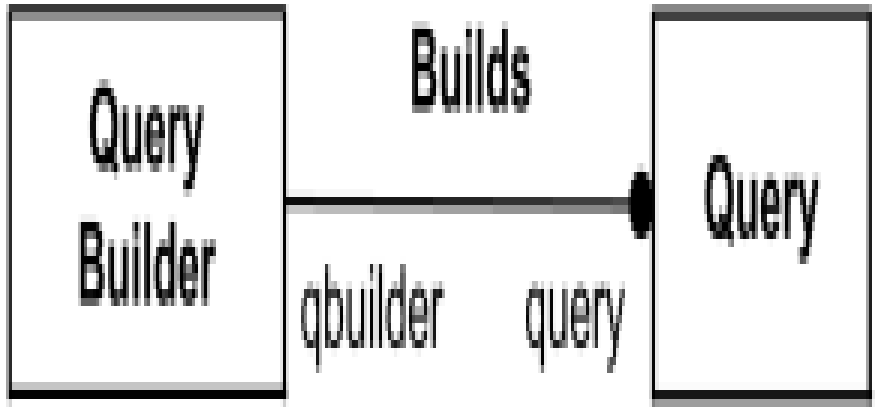
The idea of the role is that the same classifier can play the same or different roles in other associations. For example, Professor could be an author of some Books or an editor.

Association end could be owned either by

- end classifier, or
- association itself

Association ends of associations with more than two ends must be owned by the association.Ownership of association ends by an associated classifier may be indicated graphically by a small filled circle (aka dot). The dot is drawn at the point where line meets the classifier. It could be interpreted as showing that the model includes a property of the type represented by the classifier touched by the dot. This property is owned by the classifier at the other end.

# Association End




Association end query is owned by classifier QueryBuilder
and association end qbuilder is owned by association Builds itself

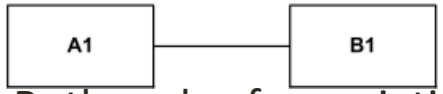
The "ownership" dot may be used in combination with the other graphic line-path notations for properties of associations and association ends. These include aggregation type and navigability.

# Navigability

**Notation**:navigable end is indicated by an open arrowhead on the end of an association

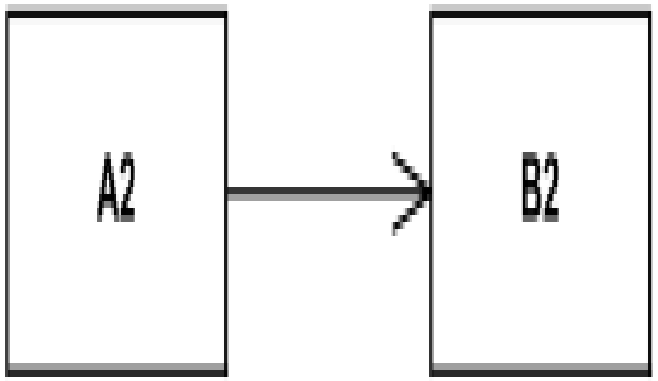
not navigable end is indicated with a small x on the end of an association

no adornment on the end of an association means unspecified navigability




Both ends of association have unspecified navigability.

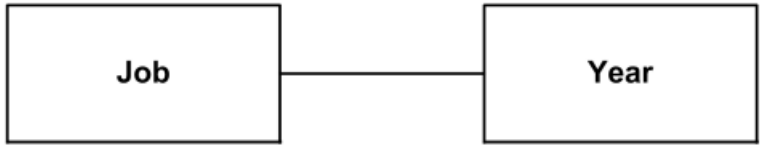
# Navigability




- A2 has unspecified navigability while B2 is navigable from A2.

# Arity

- Binary association relates two typed instances. It is normally rendered as a solid line connecting two classifiers, or a solid line connecting a single classifier to itself (the two ends are distinct). The line may consist of one or more connected segments.




Job and Year classifiers are associated

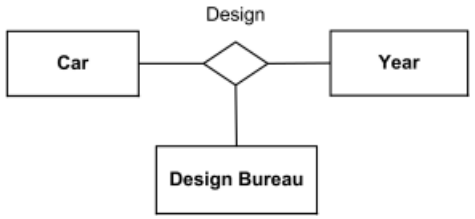
# Arity

- A small solid triangle could be placed next to or in place of the name of binary association (drawn as a solid line) to show the order of the ends of the association. The arrow points along the line in the direction of the last end in the order of the association ends. This notation also indicates that the association is to be read from the first end to the last end.


Order of the ends and reading: Car - was designed in - Year

# N-ary Association

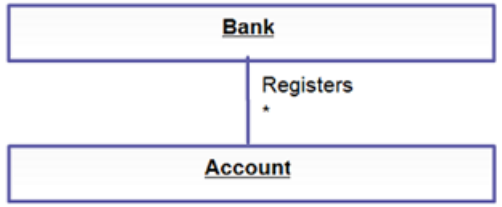
- Any association may be drawn as a diamond (larger than a terminator on a line) with a solid line for each association end connecting the diamond to the classifier that is the end's type. N-ary association with more than two ends can only be drawn this way.




Ternary association Design relates three classifiers

# Multiplicity

- The number of elements or cardinality could be defined by multiplicity. It is one of the most misunderstood relationships which describes the number of instances allowed for a particular element by providing an inclusive non-negative integers interval. It has both lower and upper bound. For example, a bank would have many accounts registered to it. Thus near the account class, a star sign is present.

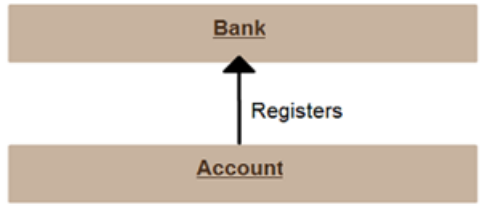


# MULTIPLICITY

| 0..1 | Zero or one |
|---|---|
| 1 | One only |
| 0..* | Zero or more |
| * | Zero or more |
| 1..* | One or more |
| 7...7 | Seven only |
| 0..1 | Zero or One |
| 4..7 | Four to seven |

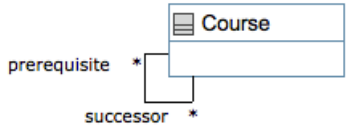
# Directed Association(NAVIGABILITY)

- This is a one-directional relationship in a class diagram which ensures the flow of control from one to another classifier. The navigability is specified by one of the association ends. The relationship between two classifiers could be described by naming any association. The direction of navigation is indicated by an arrow. Below example shows an arrowhead relationship between the container and the contained.



# Reflexive Association

- The association of a class to itself is known as Reflexive association which could be divided into Symmetric and Asymmetric type associations. In Symmetric reflexive association, the semantics of each association end has no logical difference whereas in Asymmetric Reflexive Association the associated class is the same but there is a semantic difference between the ends of the association.

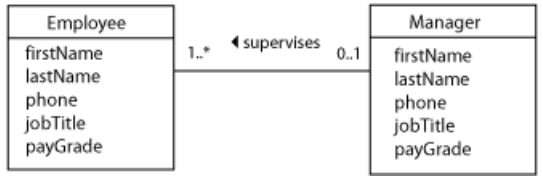


# recursive associations

A recursive association connects a single class type (serving in one role) to itself (serving in another role).

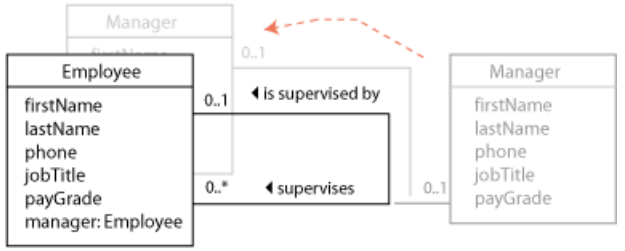
Example: In most companies, each employee (except the CEO) is supervised by one manager. Of course, not all employees are managers. This example is used in almost every database textbook, since the association between employees and managers is relatively easy to understand. Perhaps the best way to visualize it is to start with two class types:

**Incorrect model**



# correct model

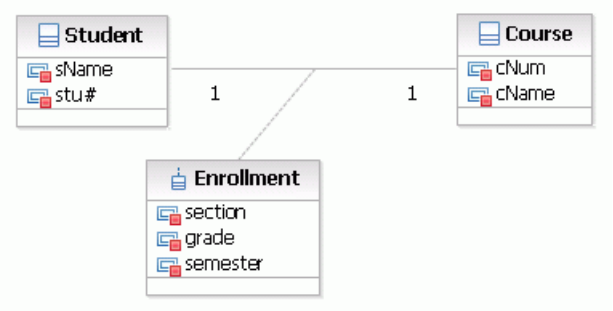
- The problem with this model is that each manager is also an employee. Building the second (manager) table not only duplicates information from the employee table, it virtually guarantees that there will be mistakes and conflicts in the data. We can fix the problem by eliminating the redundant class and re-drawing the association line.



# Association Class

An association class is a class that is part of an association relationship between two other classes.You can attach an association class to an association relationship to provide additional information about the relationship. An association class is identical to other classes and can contain operations, attributes, as well as other associations.

For example, a class called Student represents a student and has an association with a class called Course, which represents an educational course. The Student class can enroll in a course. An association class called Enrollment further defines the relationship between the Student and Course classes by providing section, grade, and semester information related to the association relationship.



# Assosiation Program

```
int main()
{driver d1,d2,d3;
car c1,c2,c3;
d1.driver(c1);
d1.driver(c3);}
d1 ->uses ->c1
d1->uses->c3
```

- Lifetime of each class does not depend on each other.
- if one class uses object of another class then it is association.

# Association

- doctor Patient
- Teacher Student
- Car Driver

A doctor can have n number of Patient .

class driver{

void drive(car &c)

{------}

}};

class car

{_____

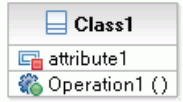
_____

}:

# Class Diagram

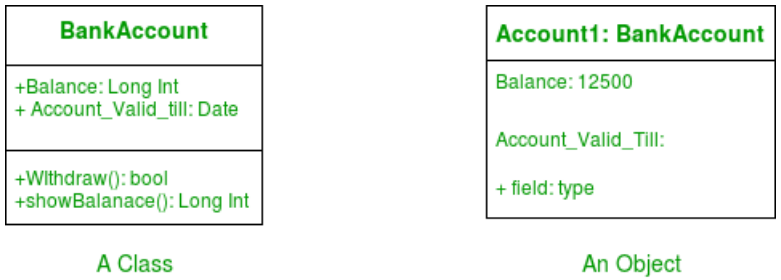
Class Diagram could be divided into three components –

The Upper Section which consists of the class name, and is a mandatory component. The middle section describes the class qualities and used while describing a class's specific instance. The bottom section describes class interaction with the data.



# Difference between an Object and a Class Diagram –

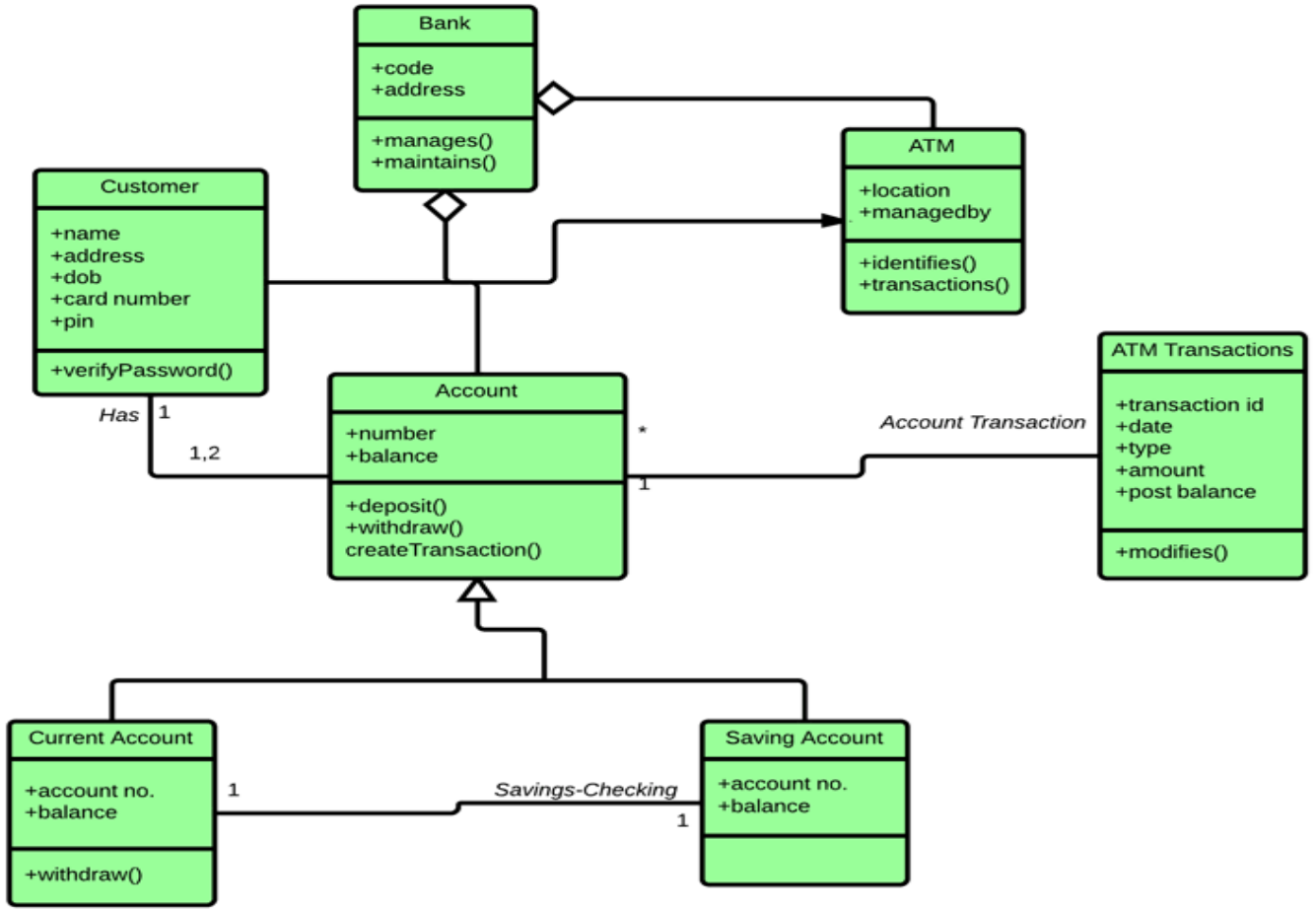
- Object Diagrams use real world examples to depict the nature and structure of the system at a particular point in time. Since we are able to use data available within objects, Object diagrams provide a clearer view of the relationships that exist between objects.



# Benefits of Class Diagram

- Class Diagram Illustrates data models for even very complex information systems
- It provides an overview of how the application is structured before studying the actual code. This can easily reduce the maintenance time
- It helps for better understanding of general schematics of an application.
- Allows drawing detailed charts which highlights code required to be programmed
- Helpful for developers and other stakeholders.

# Class Diagram



# Aggregation Syntax:

Aggregation is a way to represent HAS-A relation between the objects of 2 individual classes.. It is a subtype of association type of relation but more restrictive.

```
Class PartClass{
//instance variables
//instance methods
}
class Whole{
PartClass* partclass;
}
```

# Explanation:

- In the above syntax, the Whole class represents the class that is a container class for other Part class that is contained in the object of the whole class. Here each object of Whole class holds a reference pointer to the object of the Part class.

- For example – BUS HAS-A Engine. Here Bus is a container class. Part class is a class whose object is contained within the objects of a container class. An object of the Part class can be referred in more than 1 object of Whole class and also a lifetime of a contained class object does not depend on the lifetime of the existence of the object of a container class.

# Aggregation

- Make a pointer other class,which point to that object.

```
class department{
char name[50];
Teacher * ptr;}
public:void Tea(teacher *p)
{ptr=p;}
void  show()
{cout<<"teacher of this department is";
cout<<ptr          eid;  cout <<ptr
name;
```

→ →

```
department has id=     ; name;
int main()
{Teacher T1(201,'abc');
Department D1;
d1.Tea(&T1);
d1.show();
```

# Aggregation

Whole part relationship
Owenership

- Department ownes Teacher department forms by Teachers
- Lifetime of teacher,does not depend on department.
- person- address

```
Class Teacher{int eid;
char name[30];
public:
Teacher(int x,char  na)
eid=x;
name=na;
}
```

# Implementing aggregations

```
#include <iostream>
#include <string>
Using namespace std;
 class Teacher{
private:
  string m_name;
 public:
  Teacher(const string& name)
      : m_name{ name }
  { }
  const string& getName() {
return m_name; }};
```

# Implementing aggregations

```
class Department
{
private:
  const Teacher& m_teacher; // This dept holds only one teacher for simplicity, but it could hold many teachers

public:
  Department(const Teacher& teacher)
      : m_teacher{ teacher }
  {
  }
};
```

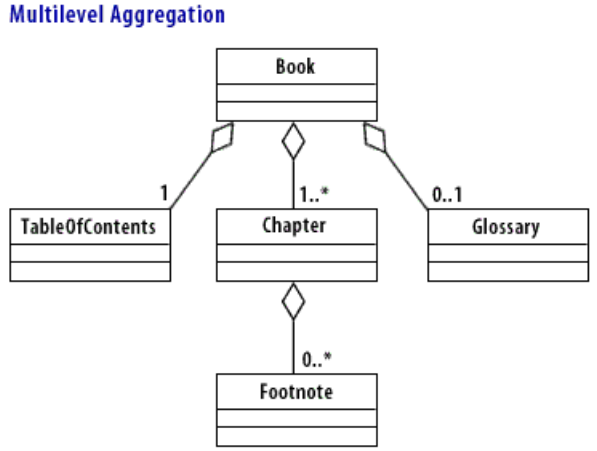
```
int main(){
  // Create a teacher outside the scope of the Department
  Teacher bob{ "Bob" }; // create a teacher

  {
    // Create a department and use the constructor parameter to pass
    // the teacher to it.
    Department department{ bob };

  } // department goes out of scope here and is destroyed

  // bob still exists here, but the department doesn't

  cout << bob.getName() << " still exists!\n";

  return 0;
}
```

In this case, bob is created independently of department, and then passed into department‘s constructor. When department is destroyed, the m_teacher reference is destroyed, but the teacher itself is not destroyed, so it still exists until it is independently destroyed later in main().

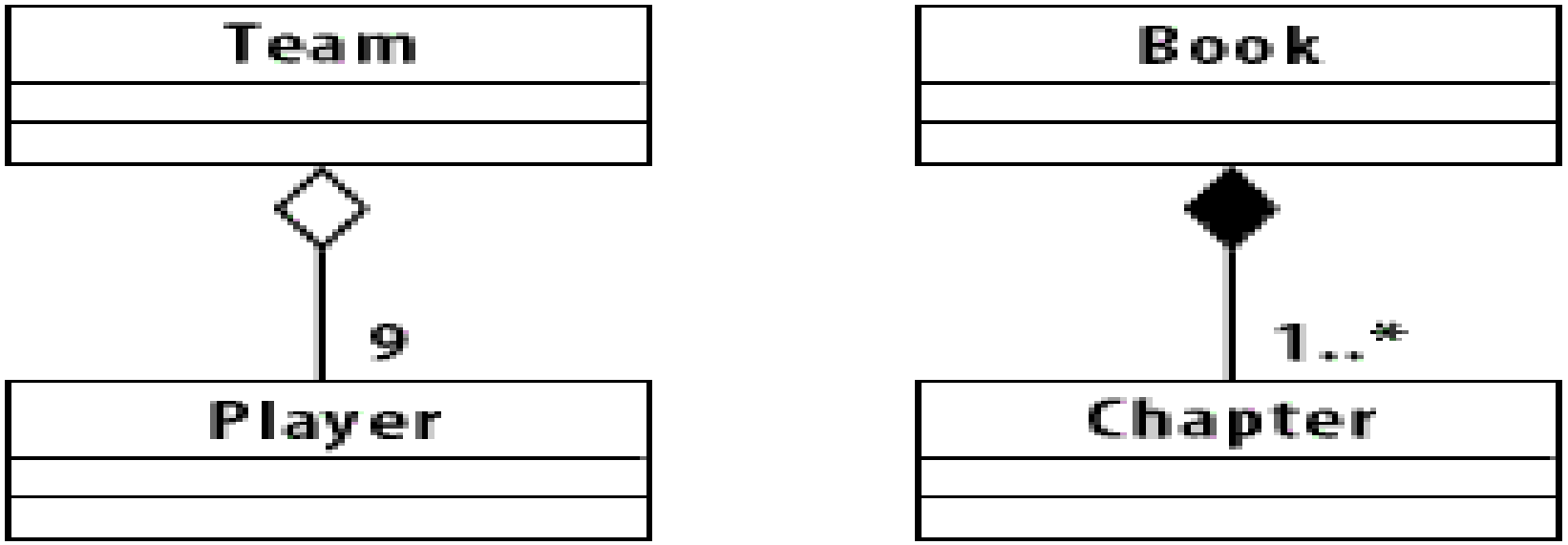
**Output-**
Bob still exists!

# Aggregation and Association Modeling

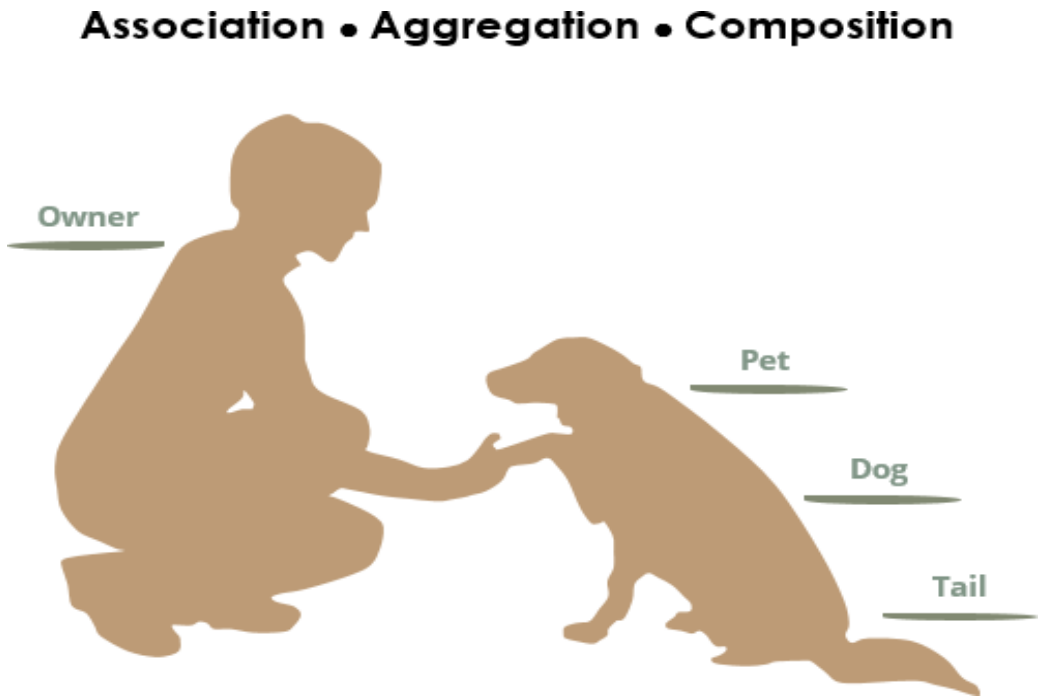
aggregation, use a diamond on the end of the association attached to the aggregate class and attach the other end of the association to the component class. Remember to assign the appropriate multiplicity to each end of the association. In most cases, the multiplicity at the aggregate end is one, so many people do not bother to set this value.



## Aggregation and Composition



# Association vs Aggregation vs Composition

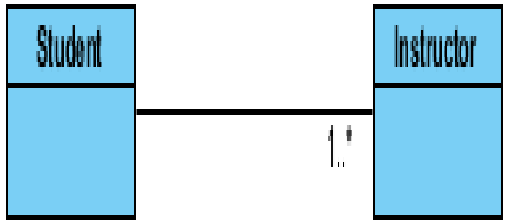



**We see the following relationships:**

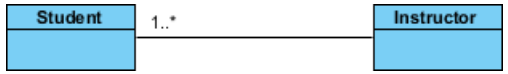
owners feed pets, pets please owners (association)
a tail is a part of both dogs and cats (aggregation / composition)
a cat is a kind of pet (inheritance / generalization)

# Association

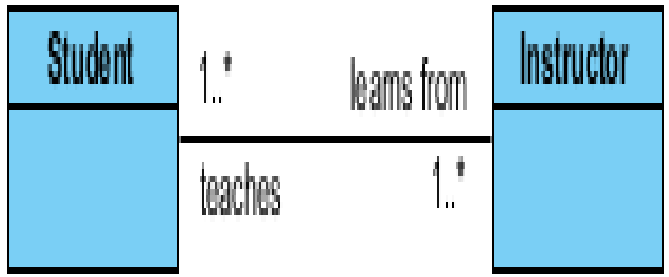
A single student can associate with multiple teachers:




The example indicates that every Instructor has one or more Students:

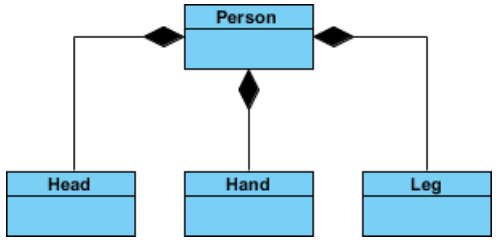



We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.



# Composition Example:

We should be more specific and use the composition link in cases where in addition to the part-of relationship between Class A and Class B - there's a strong lifecycle dependency between the two, meaning that when Class A is deleted then Class B is also deleted as a result



# Constructor Delegation in C++

- Sometimes it is useful for a constructor to be able to call another constructor of the same class. This feature, called Constructor Delegation, was introduced in C++

# An example program without delegation :

```
// A C++ program to demonstrate need of
// constructor delegation.
#include <iostream>
using namespace std;

class A {
    int x, y, z;

public:
    A()
    {
        x = 0;
        y = 0;
        z = 0;
    }
```

```
A(int z)
   {
      // The below two lines are redundant
      x = 0;
      y = 0;

      /* Only initialize z by passing an argument,
        while all the other arguments are
        initialized the same way they were,
        as in the previous constructor*/
      this->z = z;
   }

   void show()
   {
      cout << x << '\n'
           << y << '\n'
           << z;
   }
};
```

```
int main()
{
    A obj(3);
    obj.show();
    return 0;
}
```

Output:

```
0
0
3
```

# Solving above redundant code problem using constructor delegation()

```
// Program to demonstrate constructor delegation
// in C++
#include <iostream>
using namespace std;
class A {
    int x, y, z;

public:
    A()
    {
        x = 0;
        y = 0;
        z = 0;
    }
```

```
// Constructor delegation
   A(int z) : A()
   {
      this->z = z; // Only update z
   }

   void show()
   {
      cout << x << '\n'
           << y << '\n'
           << z;
   }
};
int main()
{
   A obj(3);
   obj.show();
   return 0;
}
```

**Output:**

0
0
3

It is very important to note that constructor delegation is different from calling a constructor from inside the body of another constructor, which is not recommended because doing so creates another object and initializes it, without doing anything to the object created by the constructor that called it.

# UNIT 4

**Inheritance** is one in which a new class is created that inherits the properties of the already exist class. It supports the concept of code reusability and reduces the length of the code in object-oriented programming.

Types of Inheritance are:

- Single inheritance
- Multi-level inheritance
- Multiple inheritance
- Hybrid inheritance
- Hierarchical inheritance

# Example of Inheritance:

```
#include "iostream"
using namespace std;
  class A {  int a, b;
public:
 void add(int x, int y)  {
        a = x;
        b = y;
        cout << (a + b) << endl; } };
  class B : public A {
    public:
    void print(int x, int y)
    {  add(x, y); } };
```

```
int main()
{
   B b1;
   b1.print(5, 6);
}
```

**OUTPUT**-addition of a+b is:11

Here, class B is the derived class which inherit the property(add method) of the base class A.

# Polymorphism

**Polymorphism** is that in which we can perform a task in multiple forms or ways. It is applied to the functions or methods. Polymorphism allows the object to decide which form of the function to implement at compile-time as well as run-time.

Types of Polymorphism are:

- Compile-time polymorphism (Method overloading)
- Run-time polymorphism (Method Overriding)

# Example of Polymorphism:

```
#include "iostream"
using namespace std;
 class A { int a, b, c;
 public:
 void add(int x, int y)
  {  a = x;
     b = y;
     cout << "addition of a+b is:" << (a + b) << endl;   }
 void add(int x, int y, int z)
  {
     a = x;
     b = y;
     c = z;
     cout << "addition of a+b+c is:" << (a + b + c) << endl;   }
```

```
void print()
   {
      cout << "Class A's method is running" << endl;
   }
};

class B : public A {
public:
   void print()
   {
      cout << "Class B's method is running" << endl;
   }
};
int main()
{ A a1;
   // method overloading (Compile-time polymorphism)
a1.add(6, 5);
 // method overloading (Compile-time polymorphism)
   a1.add(1, 2, 3);
 B b1;
  // Method overriding (Run-time polymorphism)
   b1.print();
}
```

Output:

addition of a+b is:11
addition of a+b+c is:6
Class B's method is running

# Difference between Inheritance and Polymorphism:

| S.NO | INHERITANCE | POLYMORPHISM |
|---|---|---|
| 1. | Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class). | Whereas polymorphism is that which can be defined in multiple forms. |
| 2. | It is basically applied to classes. | Whereas it is basically applied to functions or methods. |
| 3. | Inheritance supports the concept of reusability and reduces code length in object-oriented programming. | Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding). |
| 4. | Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance. | Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding). |
| 5. | It is used in pattern designing. | While it is also used in pattern designing. |

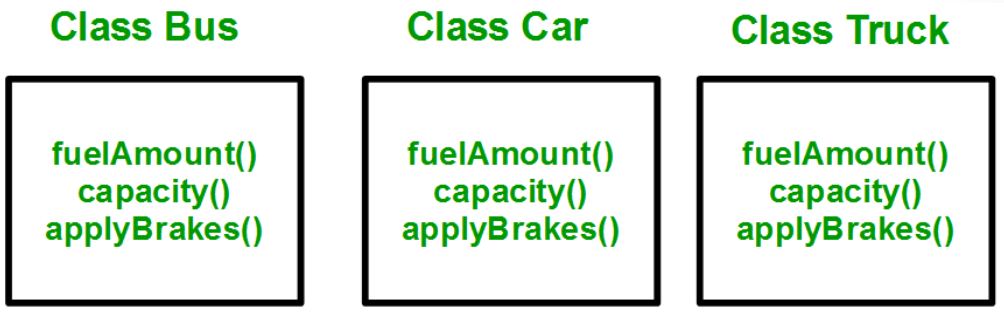
# Inheritance in C++

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature 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.

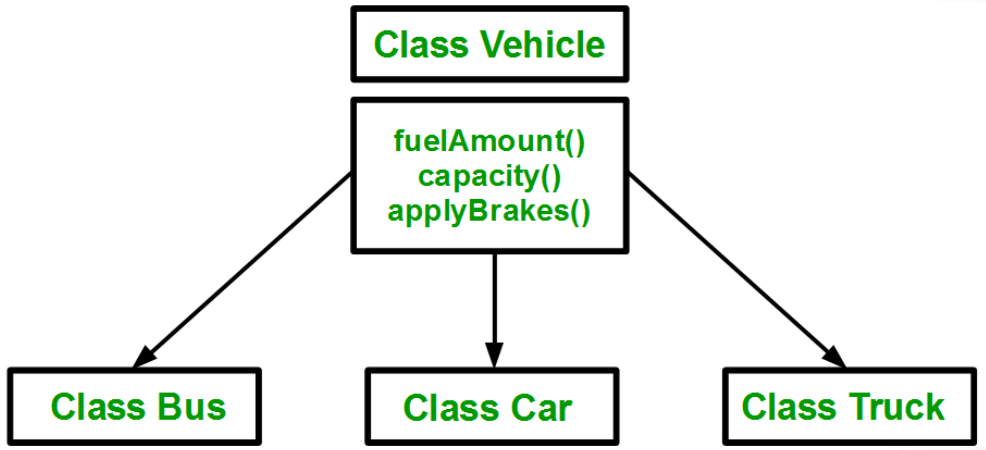
# Why and when to use inheritance?

- Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



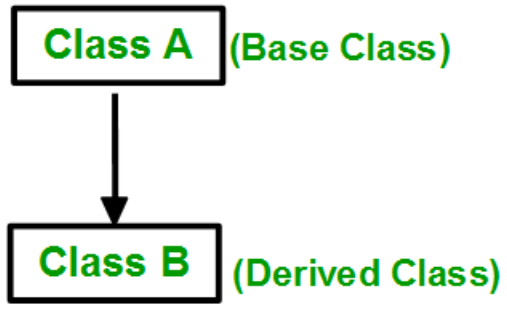
# Why and when to use inheritance?

- You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



# Types of Inheritance in C++

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



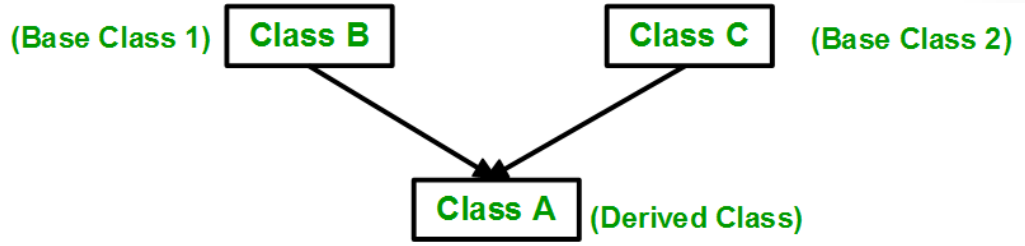
```
#include <iostream>
using namespace std;
  // base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;  }
};

// sub class derived from two base classes
class Car: public Vehicle{
  };
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

This is a vehicle

# Multiple Inheritance:

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.




**Syntax:**

```
class subclass_name : access_mode base_class1,
access_mode base_class2, ....
{
  //body of subclass
};
```

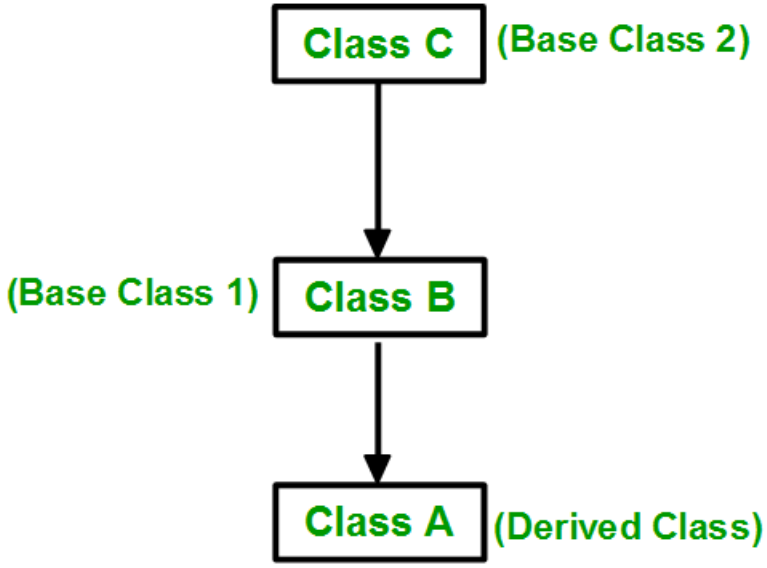
```
#include <iostream>
using namespace std;
  // first base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl; } };
  // second base class
class FourWheeler {
  public:
    FourWheeler()
    { cout << "This is a 4 wheeler Vehicle" << endl;  } };
  // sub class derived from two base classes
class Car: public Vehicle, public FourWheeler { };
// main function
int main()
{     // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;}
```

**Output:**

This is a Vehicle
This is a 4 wheeler Vehicle

# Multilevel Inheritance:

In this type of inheritance, a derived class is created from another derived class.



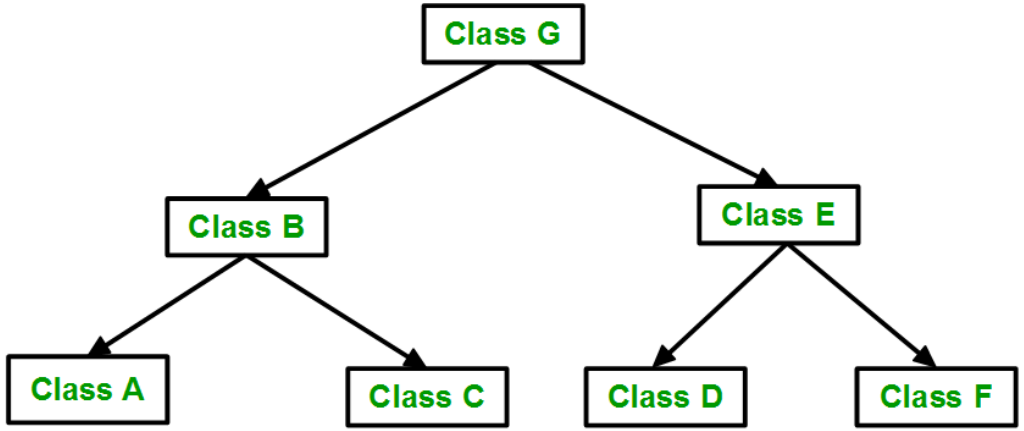
```
 class Vehicle           // base class
{ public:
   Vehicle()
   {
     cout << "This is a Vehicle" << endl;
   } };
class fourWheeler: public Vehicle
{  public:
   fourWheeler()
   {
     cout<<"Objects with 4 wheels are vehicles"<<endl;
   } }; // sub class derived from two base classes
class Car: public fourWheeler{
  public:
    car()
    {
      cout<<"Car has 4 Wheels"<<endl; } };
 // main function
int main()
{    //creating object of sub class will
   //invoke the constructor of base classes
   Car obj;
   return 0;
}
```

**output:**

```
This is a Vehicle
Objects with 4 wheels
are vehicles
Car has 4 Wheels
```

# Hierarchical Inheritance:

In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



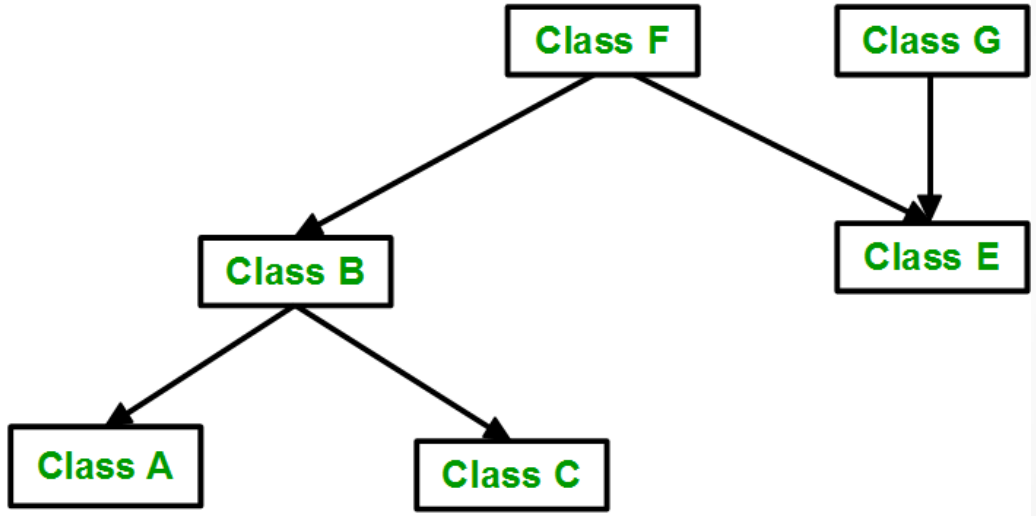
```
#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;  } };
  // first sub class
class Car: public Vehicle
{ };
// second sub class
class Bus: public Vehicle
{ };
int main()
{    // creating object of sub class will
    // invoke the constructor of base class
    Car obj1;
    Bus obj2;
    return 0; }
```

Output:

```
This is a Vehicle
This is a Vehicle
```

# Hybrid (Virtual) Inheritance:

Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.Below image shows the combination of hierarchical and multiple inheritance:



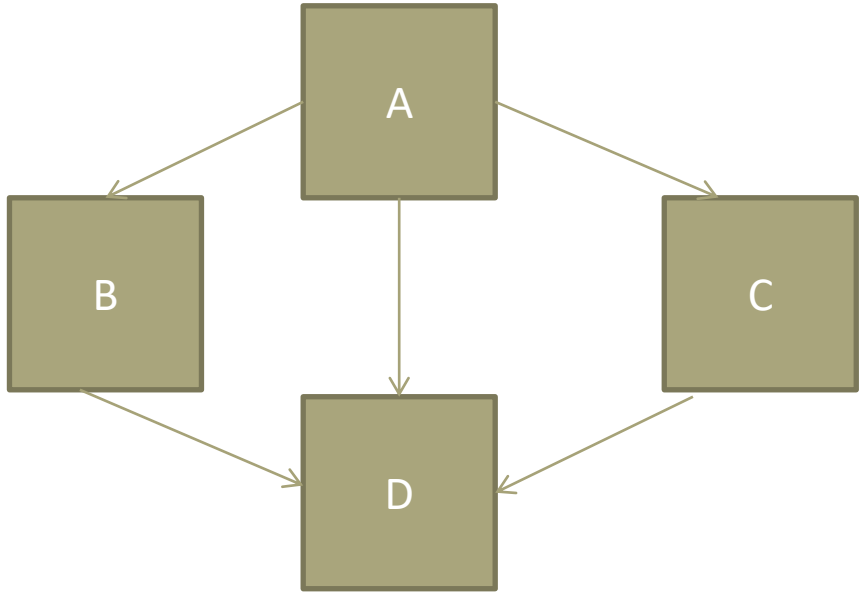
```
class Vehicle
{  public:
    Vehicle()
    { cout << "This is a Vehicle" << endl; } };
  //base class
class Fare
{ public:
    Fare()
    { cout<<"Fare of Vehicle\n";  } };
  // first sub class
class Car: public Vehicle
{ }; // second sub class
class Bus: public Vehicle, public Fare
{ };
  // main function
int main()
{    // creating object of sub class will
    // invoke the constructor of base class
    Bus obj2;
    return 0;
}
```

Output:

This is a Vehicle
Fare of Vehicle

# A special case of hybrid inheritance : Multipath inheritance:

- A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. An ambiguity can arrise in this type of inheritance.



# Consider the following program:

```
#include<iostream.h>
class ClassA {
 public:
          int a;  };
class ClassB : public ClassA
 {  public:
          int b;   };
  class ClassC : public ClassA {
          public:
          int c;   };
  class ClassD : public ClassB, public ClassC
      {    public:   int d;
      };
```

```
void main()
    {  ClassD obj;
  //obj.a = 10;              //Statement 1, Error
          //obj.a = 100;             //Statement 2, Error
  obj.ClassB::a = 10;       //Statement 3
          obj.ClassC::a = 100;     //Statement 4
  obj.b = 20; obj.c = 30;  obj.d = 40;
  cout<< "\n A from ClassB  : "<< obj.ClassB::a;
   cout<< "\n A from ClassC  : "<< obj.ClassC::a;
  cout<< "\n B : "<< obj.b;
    cout<< "\n C : "<< obj.c;
     cout<< "\n D : "<< obj.d;
 }
```

Output:

A from ClassB  : 10
A from ClassC  : 100
B : 20
C : 30
D : 40

# A special case of hybrid inheritance : Multipath inheritance:

In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA. However ClassD inherit both ClassB & ClassC, therefore ClassD have two copies of ClassA, one from ClassB and another from ClassC.

If we need to access the data member a of ClassA through the object of ClassD, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bco'z compiler can't differentiate between two copies of ClassA in ClassD.

There are 2 ways to avoid this ambiguity:

- Use scope resolution operator
- Use virtual base class

# Avoiding ambiguity using scope resolution operator:

Using scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statement 3 and 4, in the above example.

- obj.ClassB::a = 10;  //Statement 3
- obj.ClassC::a = 100;  //Statement 4

Note : Still, there are two copies of ClassA in ClassD.

# Avoiding ambiguity using virtual base class:

```
#include<iostream.h>
 class ClassA  {
     public:
          int a;   };
 class ClassB : virtual public ClassA
    {  public:
          int b; };
    class ClassC : virtual public ClassA
    { public:
          int c; };
class ClassD : public ClassB, public ClassC
    { public:
          int d;   };
```

```
void main()
   {

        ClassD obj;

        obj.a = 10;       //Statement 3
        obj.a = 100;      //Statement 4

        obj.b = 20;
        obj.c = 30;
        obj.d = 40;

        cout<< "\n A : "<< obj.a;
        cout<< "\n B : "<< obj.b;
        cout<< "\n C : "<< obj.c;
        cout<< "\n D : "<< obj.d;

   }
```

**Output:**

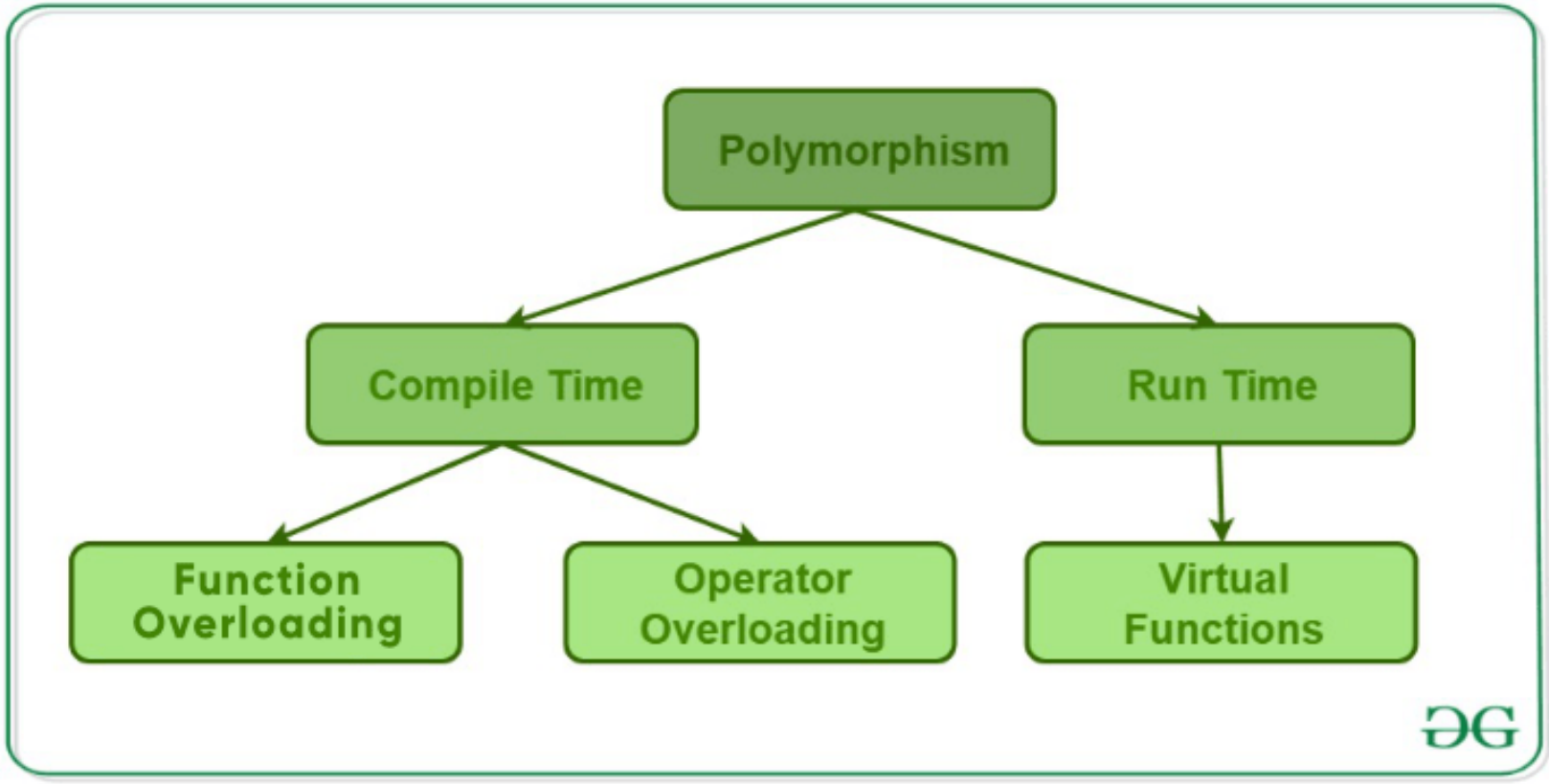
A : 100
B : 20
C : 30
D : 40

According to the above example, ClassD has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given at statement 3.

# Polymorphism in C++

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. A real-life example 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 posses different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

# In C++ polymorphism is mainly divided into two types:



# Compile time polymorphism:

This type of polymorphism is achieved by function overloading or operator overloading.

**Function overloading-**When there are multiple functions with same name but different parameters then these **functions are said to be overloaded**. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

# Function overloading-

```
#include <bits/stdc++.h>
  using namespace std;
class Geeks
{   public:
      // function with 1 int parameter
    void func(int x)
    {  cout << "value of x is " << x << endl;  }
      // function with same name but 1 double parameter
    void func(double x)
    { cout << "value of x is " << x << endl;  }
 // function with same name and 2 int parameters
    void func(int x, int y)
    { cout << "value of x and y is " << x << ", " << y << endl;   } };
```

```
int main() {

    Geeks obj1;

    // Which function is called will depend on the parameters
passed
    // The first 'func' is called
    obj1.func(7);

    // The second 'func' is called
    obj1.func(9.132);

    // The third 'func' is called
    obj1.func(85,64);
    return 0;
}
```

**Output:**

value of x is 7
value of x is 9.132
value of x and y is 85, 64

In the above example, a single function named func acts differently in three different situations which is the property of polymorphism.

# operator overloading.

- C++ also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator ‘+’ when placed between integer operands , adds them and when placed between string operands, concatenates them.

```
#include<iostream>
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

In the above example the operator ‘+’ is overloaded. The operator ‘+’ is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers.

# Run Time Polymorphism

This type of polymorphism is achieved by Function Overriding.

**Function overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

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

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

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

```
class derived:public base
{
public:
   void print () //print () is already virtual
function in derived class, we could also
declared as virtual void print () explicitly
   { cout<< "print derived class" <<endl; }

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

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

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

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

    return 0;
}
```

**Output:**

print derived class
show base class

Consider the following simple program as an example of runtime polymorphism. The main thing to note about the program is that the derived class's function is called using a base class pointer.

The idea is that virtual functions are called according to the type of the object instance pointed to or referenced, not according to the type of the pointer or reference.

In other words, virtual functions are resolved late, at runtime.

```
#include <iostream>
using namespace std;

class Base {
public:
   virtual void show()
   {
      cout << " In Base \n";
   }
};

class Derived : public Base {
public:
   void show()
   {
      cout << "In Derived \n";
   } };
```

```
int main(void)
{
   Base* bp = new
Derived;

   // RUN-TIME
POLYMORPHISM
   bp->show();

   return 0;
}
```

**Output:**

In Derived

# Access Modifiers and Inheritance: Visibility of Class Members

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

**Public Inheritance:**

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public

class Subclass : public Superclass

# Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

class Subclass : Superclass  // By default its private inheritance

# Protected Inheritance

In protected mode, the public and protected members of Super class becomes protected members of Sub class.

class subclass : protected Superclass

# Table showing all the Visibility Modes

| | Derived Class | Derived Class | Derived Class |
|---|---|---|---|
| **Base class** | **Public Mode** | **Private Mode** | **Protected Mode** |
| **Private** | Not Inherited | Not Inherited | Not Inherited |
| **Protected** | Protected | Private | Protected |
| **Public** | Public | Private | Protected |

# Operators Overloading in C++

- You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.
- Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

**Box operator+(const Box&);**

declares the **addition operator** that can be used to add two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary **non-member functions or as class member functions**. In case we define above function as **non-member function of a class** then we would have to pass two arguments for each operand as follows −

**Box operator+(const Box&, const Box&);**

# Overloadable/Non-overloadableOperators

- Following is the list of operators which can be overloaded –

| + | - | * | / | % | ^ |
|---|---|---|---|---|---|
| & | \| | ~ | ! | , | = |
| < | > | <= | >= | ++ | -- |
| << | >> | == | != | && | \|\| |
| += | -= | /= | %= | ^= | &= |
| \|= | *= | <<= | >>= | [] | () |
| -> | ->* | new | new [] | delete | delete [] |

# Following is the list of operators, which can not be overloaded –

| | | | |
|---|---|---|---|
| :: | .* | . | ?: |

# Interfaces in C++ (Abstract Classes)

- An interface describes the behavior or capabilities of a C++ class **without committing to a particular implementation of that class.**

- The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is **a concept of keeping implementation details separate from associated data.**

- A class is made abstract by declaring at least one of its functions as pure virtual function. **A pure virtual function is specified by placing "= 0" in its declaration as follows –**

# Interfaces in C++ (Abstract Classes)

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

# Interfaces in C++ (Abstract Classes)

- The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. **Abstract classes cannot be used to instantiate objects** and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.

- Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. **Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.**

- **Classes that can be used to instantiate objects are called concrete classes.**

# Abstract Class Example

- Consider the following example where parent class provides an interface to the base class to implement a function called getArea() −

```
class Shape {
  public:
     virtual int getArea() = 0;
    void setWidth(int w)
{width = w;  }
  void setHeight(int h) {
      height = h;  }
   protected:
int width;
    int height;};
```

# Abstract Class Example

```
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height); }};
class Triangle: public Shape {
  public:
    int getArea() {
      return (width * height)/2; }};
```

# Abstract Class Example

```
int main(void) {
Rectangle Rect;
   Triangle  Tri;
  Rect.setWidth(5);
   Rect.setHeight(7);
      cout << "Total Rectangle area: " << Rect.getArea() << endl;
 Tri.setWidth(5);
   Tri.setHeight(7);
     cout << "Total Triangle area: " << Tri.getArea() << endl;
return 0;}
```

**OUTPUT-**

Total Rectangle area: 35

Total Triangle area: 17

You can see how an abstract class defined an interface in terms of getArea() and two other classes implemented same function but with different algorithm to calculate the area specific to the shape.

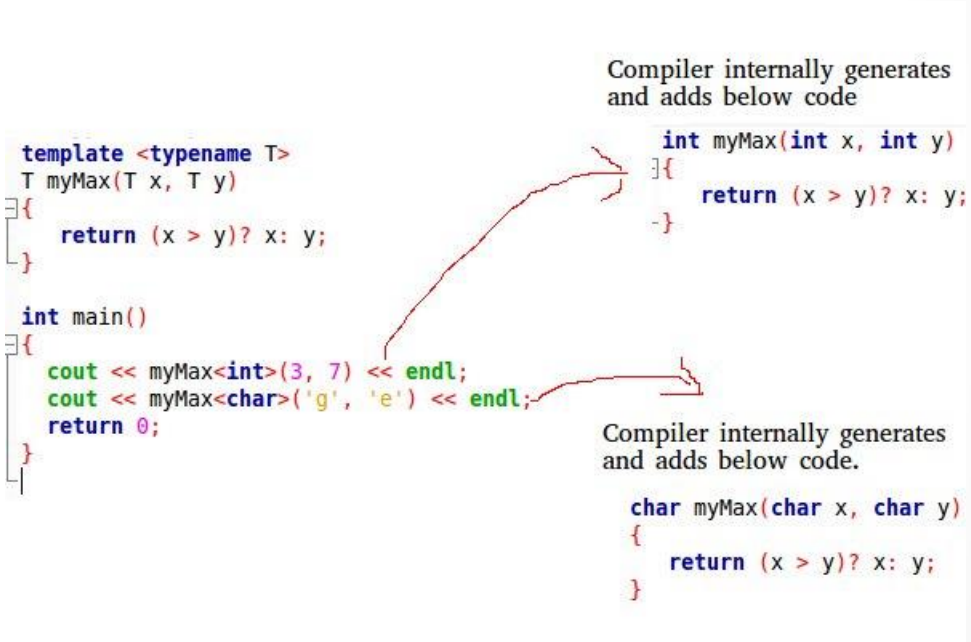
# UNIT 5

**Templates in C++**

A template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don’t need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

# How templates work?

- Templates are expanded at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.



# Function Template

The general form of a template function definition is shown here -

```
template <class type> ret-type func-name(parameter list) {
  // body of function
}
```

A function template starts with the keyword template followed by template parameter/s inside < > which is followed by function declaration.

```
template <class T>
T someFunction(T arg)
{
  … .. …
}
```

In the above code, T is a template argument that accepts different data types (int, float), and class is a keyword.

You can also use keyword typename instead of class in the above example.

When, an argument of a data type is passed to someFunction( ), compiler generates a new version of someFunction() for the given data type.

# Function Templates

We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().

```
#include <iostream>
using namespace std;
// One function works for all data types.  This would work
// even for user defined types if operator '>' is overloaded
template <typename T>
T myMax(T x, T y)
{   return (x > y)? x: y; }
  int main()
{ cout << myMax<int>(3, 7) << endl;  // Call myMax for int
  cout << myMax<double>(3.0, 7.0) << endl; // call myMax for
double
  cout << myMax<char>('g', 'e') << endl;   // call myMax for char
  return 0; }
```

**Output:**

```
7
7
g
```

# Class Template

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here –

```
template <class type> class class-name {
   .
   .
   .
}
```

Here, type is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

# Class Template

How to declare a class template?

```
template <class T>
class className
{
   … .. …
public:
   T var;
   T someOperation(T arg);
   … .. …
};
```

In the above declaration, T is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable var and a member function someOperation() are both of type T.

# How to create a class template object?

To create a class template object, you need to define the data type inside a < > when creation.

className<dataType> classObject;

For example:

className<int> classObject;

className<float> classObject;

className<string> classObject;

# Simple calculator using Class template

```
template <class T>
class Calculator{
private:
T num1, num2;
public:
Calculator(T n1, T n2){
                    num1 = n1;
                    num2 = n2;}
```

```
void displayResult()
        {
                cout << "Numbers are: " << num1 << "
and " << num2 << "." << endl;
                cout << "Addition is: " << add() << endl;
                cout << "Subtraction is: " << subtract() <<
endl;
                cout << "Product is: " << multiply() <<
endl;
                cout << "Division is: " << divide() << endl;
        }

        T add() { return num1 + num2; }

        T subtract() { return num1 - num2; }

        T multiply() { return num1 * num2; }

        T divide() { return num1 / num2; }
};
```

```
int main()
{
        Calculator<int> intCalc(2, 1);
        Calculator<float> floatCalc(2.4, 1.2);

        cout << "Int results:" << endl;
        intCalc.displayResult();

        cout << endl << "Float results:" << endl;
        floatCalc.displayResult();

        return 0;
}
```

Output

```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and
1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

# Containers in C++ STL (Standard Template Library)

- A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

- The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers)

# Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

- array: Static contiguous array (class template)
- vector: Dynamic contiguous array (class template)
- deque: Double-ended queue (class template)
- forward_list: Singly-linked list (class template)
- list : Doubly-linked list (class template)

# Associative containers

Associative containers implement sorted data structures that can be quickly searched (O(log n) complexity).

- Set: Collection of unique keys, sorted by keys(class template)
- Map: Collection of key-value pairs, sorted by keys, keys are unique (class template).
- multiset: Collection of keys, sorted by keys (class template)
- multimap: Collection of key-value pairs, sorted by keys(class template)

# Unordered associative containers

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (O(1) amortized, O(n) worst-case complexity).

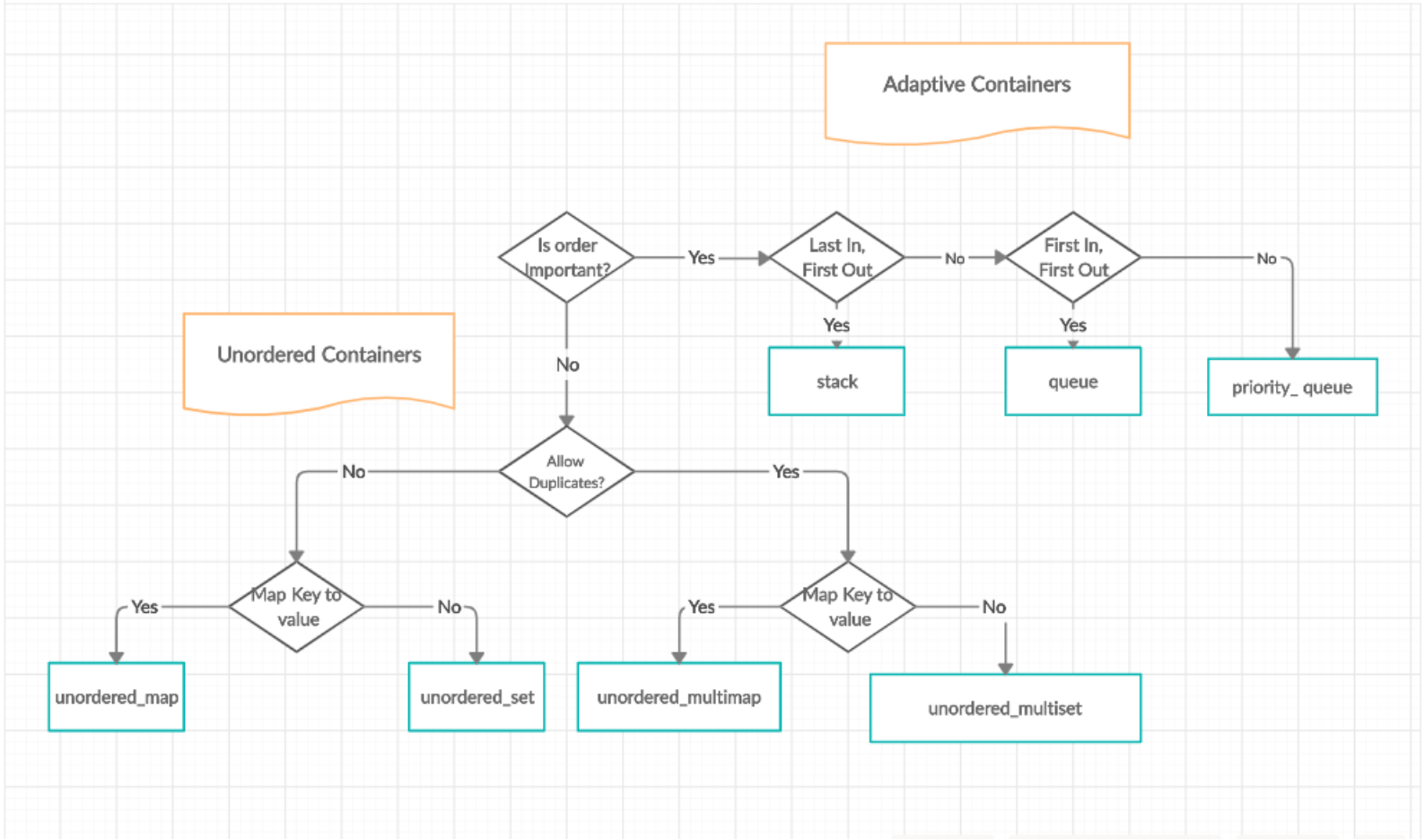
- unordered_set: Collection of unique keys, hashed by keys. (class template)
- unordered_map: Collection of key-value pairs, hashed by keys, keys are unique. (class template)
- unordered_multiset: Collection of keys, hashed by keys (class template)
- unordered_multimap: Collection of key-value pairs, hashed by keys (class template)

# Container adaptors

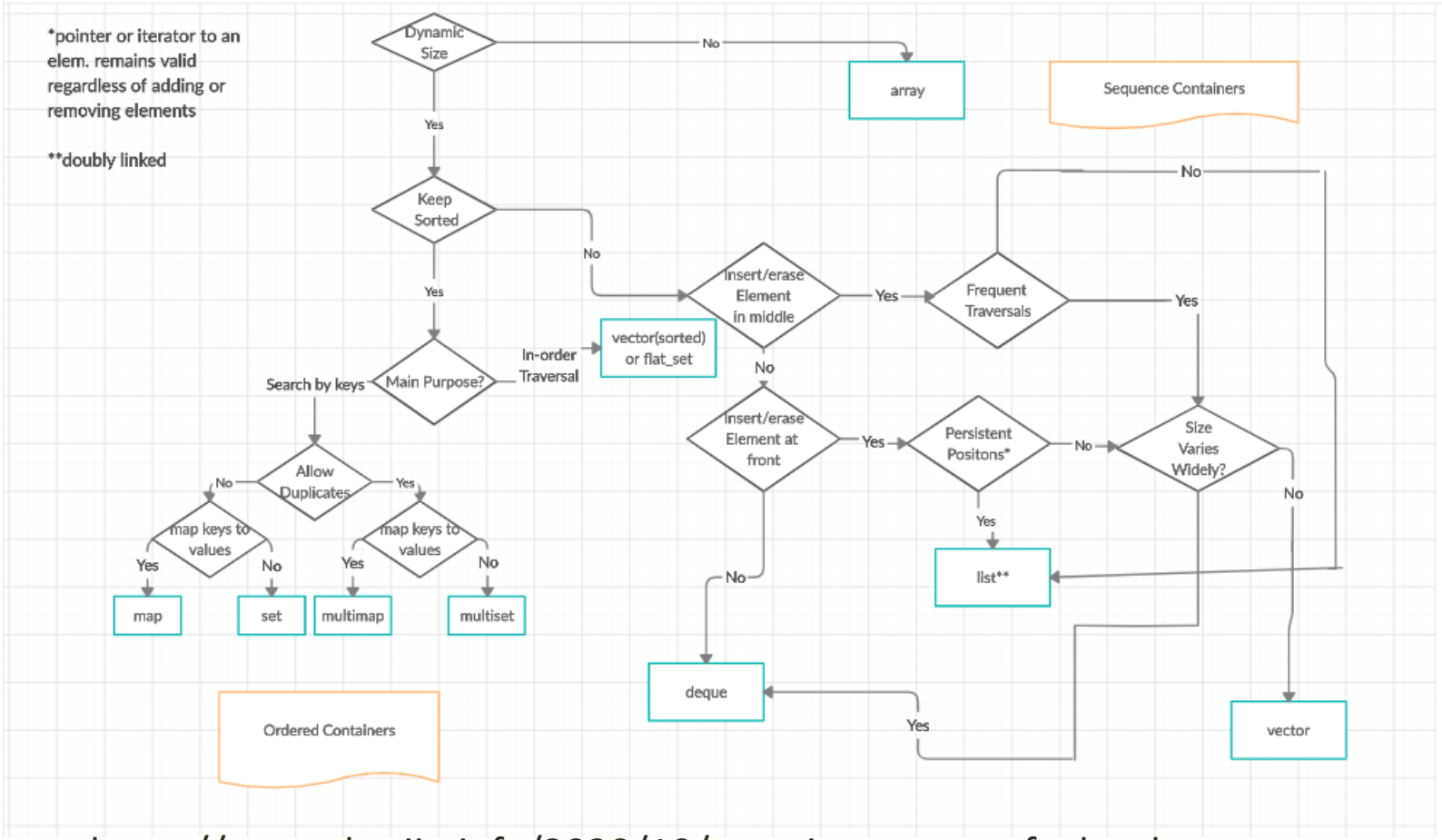
Container adaptors **provide a different interface for sequential containers.**

- stack: Adapts a container to provide stack (LIFO data structure) (class template).
- queue: Adapts a container to provide queue (FIFO data structure) (class template).
- priority_queue: Adapts a container to provide priority queue (class template).

# Flowchart of Adaptive Containers and Unordered Containers



# Flowchart of Sequence conatiners and ordered containers




https://www.sherijo.info/2020/10/container-types-of-c.html

# Iterators in C++ STL

Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequence of numbers, characters etc. They reduce the complexity and execution time of program.

Operations of iterators :-

- 1. begin() :- This function is used to return the beginning position of the container.
- 2. end() :- This function is used to return the after end position of the container.

# EXAMPLE

```
// C++ code to demonstrate the working of // iterator, begin() and end()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{  vector<int> ar = { 1, 2, 3, 4, 5 };
     // Declaring iterator to a vector
   vector<int>::iterator ptr;
     // Displaying vector elements using begin() and end()
   cout << "The vector elements are : ";
   for (ptr = ar.begin(); ptr < ar.end(); ptr++)
      cout << *ptr << " ";
  return 0;     }
```

**OUPUT-**

The vector elements are
: 1 2 3 4 5

# Iterators in C++ STL

- **3. advance()** :- This function is used to increment the iterator position till the specified number mentioned in its arguments.

```
// C++ code to demonstrate the working of // advance()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{ vector<int> ar = { 1, 2, 3, 4, 5 };
    // Declaring iterator to a vector
   vector<int>::iterator ptr = ar.begin();
     // Using advance() to increment iterator position
   // points to 4
   advance(ptr, 3);
     // Displaying iterator position
   cout << "The position of iterator after advancing is : ";
   cout << *ptr << " ";
       return 0; }
```

**OUTPUT-**

The position of iterator after advancing is : 4

# Iterators in C++ STL

- 4next() :- This function returns the new iterator that the iterator would point after advancing the positions mentioned in its arguments.

- 5. prev() :- This function returns the new iterator that the iterator would point after decrementing the positions mentioned in its arguments.

```
// C++ code to demonstrate the working of
// next() and prev()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
   vector<int> ar = { 1, 2, 3, 4, 5 };  // Declaring iterators to a vector
   vector<int>::iterator ptr = ar.begin();
   vector<int>::iterator ftr = ar.end();  // Using next() to return new iterator points to 4
   auto it = next(ptr, 3); // Using prev() to return new iterator  points to 3
   auto it1 = prev(ftr, 3); // Displaying iterator position
   cout << "The position of new iterator using next() is : ";
   cout << *it << " ";
   cout << endl; // Displaying iterator position
   cout << "The position of new iterator using prev()  is : ";
   cout << *it1 << " ";
   cout << endl;

   return 0;
}
```

**OUTPUT-The position of new iterator using next() is : 4**
**The position of new iterator using prev()  is : 3**

**6. inserter()** :- This function is used to insert the elements at any position in the container. It accepts 2 arguments, the container and iterator to position where the elements have to be inserted.

```
// C++ code to demonstrate the working of inserter()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{ vector<int> ar = { 1, 2, 3, 4, 5 };
   vector<int> ar1 = {10, 20, 30};  // Declaring iterator to a vector
   vector<int>::iterator ptr = ar.begin(); // Using advance to set position
   advance(ptr, 3); // copying 1 vector elements in other using inserter() inserts ar1 after 3rd position
in ar
   copy(ar1.begin(), ar1.end(), inserter(ar,ptr));  // Displaying new vector elements
   cout << "The new vector after inserting elements is : ";
   for (int &x : ar)  //for(int x : temp) is a c++11 style loop that just runs over (iterates) each element in
the vector and copies it into x.
       cout << x << " ";
      return 0;     }
```

**Output:**

**The new vector after inserting elements is : 1 2 3 10 20 30 4 5**

# Heterogeneous containers

Heterogenous containers, as opposed to regular homogenous containers, are containers containing different types; that is, in homogenous containers, such as std::vector, std::list, std::set, and so on, every element is of the same type. A heterogeneous container is a container where elements may have different types.

- **Static-sized heterogenous containers**

C++ comes with two heterogeneous containers, **std::pair** and **std::tuple.** As std::pair is a subset of std::tuple with only two elements, we will only focus on std::tuple.

# The std::tuple container

**The std::tuple** is a statically sized heterogeneous container that can be declared to be of any size. In contrast to std::vector, for example, its size cannot change at runtime; you cannot add or remove elements.

A tuple is constructed with its member types explicitly declared like this:

```
auto tuple0 = std::tuple<int, std::string, bool>{};
```

# Member functions

| **CONSTRUCTOR** | **constructs a new tuple (public member function)** |
| --- | --- |
| operator | assigns the contents of one tuple to another (public member function) |
| swap | swaps the contents of two tuples |

# Non-member functions

| **make _tuple** | **creates a tuple object of the type defined by the argument types<br>(function template)** |
|---|---|
| tie | creates a tuple of lvalue references or unpacks a tuple into individual objects<br>(function template) |
| forward_as_tuple | creates a tuple of forwarding references<br>(function template) |
| tuple_cat | creates a tuple by concatenating any number of tuples<br>(function template)<br>std::get(std::tuple) |

```
#include <tuple>
#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;
tuple<double, char, string> get_student(int id)
{if (id == 0) return make_tuple(3.8, 'A', "Lisa Simpson");
   if (id == 1) return make_tuple(2.9, 'C', "Milhouse Van Houten");
   if (id == 2) return make_tuple(1.7, 'D', "Ralph Wiggum");
   throw invalid_argument("id");}
// C++17 structured binding:
int main()
{  auto student0 = get_student(0);
   std::cout << "ID: 0, "
             << "GPA: " << get<0>(student0) << ", "
             << "grade: " << get<1>(student0) << ", "
             << "name: " << get<2>(student0) << '\n';
```

```
  auto student1 = get_student(1);
   cout << "ID: 1, "
        << "GPA: " << get<0>(student1) << ", "
        << "grade: " << get<1>(student1) << ", "
        << "name: " << get<2>(student1) << '\n';

auto student2 = get_student(2);
   cout << "ID: 2, "
        << "GPA: " << get<0>(student2) << ", "
        << "grade: " << get<1>(student2) << ", "
        << "name: " << get<2>(student2) << '\n';

 }
```

```
ID: 0, GPA: 3.8, grade: A,
name: Lisa Simpson
ID: 1, GPA: 2.9, grade: C,
name: Milhouse Van
Houten
ID: 2, GPA: 1.7, grade: D,
name: Ralph Wiggum
```

# file and stream in c++

- How to read and write from a file. This requires another standard C++ library called fstream, which defines three new data types −

| Sr.No | Data Type & Description |
|---|---|
| 1 | ofstream<br>This data type represents the output file stream and is used to create files and to write information to files. |
| 2 | ifstream<br>This data type represents the input file stream and is used to read information from files. |
| 3 | fstream<br>This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files. |

# file and stream in c++

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

**Opening a File**

A file must be opened before you can read from it or write to it. Either ofstream or fstream object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

void open(const char *filename, ios::openmode mode);

Here, the first argument specifies the name and location of the file to be opened and the second argument of the open() member function defines the mode in which the file should be opened.

| Sr.No | Mode Flag & Description |
|---|---|
| 1 | ios::app<br>Append mode. All output to that file to be appended to the end. |
| 2 | ios::ate<br>open a file in this mode for output and read/write controlling to the end of the file. |
| 3 | ios::in<br>Open a file for reading. |
| 4 | ios::out<br>Open a file for writing. |
| 5 | ios::trunc<br>If the file already exists, its contents will be truncated before opening the file. |

combine two or more of these values by ORing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax −

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows −

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

**Closing a File**

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

**Writing to a File**

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the cout object.

**Reading from a File**

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an ifstream or fstream object instead of the cin object.

# Functions for file input and output

- open(): To create a file
- close(): To close an existing file
- get(): to read a single character from the file
- put(): to write a single character in the file
- read(): to read data from a file
- write(): to write data into a file

# Reading from and Writing to a file

While doing C++ program, programmers write information to a file from the program using the stream insertion operator (<<) and reads information using the stream extraction operator (>>). The only difference is that for files programmers need to use an ofstream or fstream object instead of the cout object and ifstream or fstream object instead of the cin object.

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
 { fstream file;
  file.open("egone.txt",ios::out);
 cout<< "Writing to a file in C++....";
 file.close();
  return 0;
}
```

# Persistent Object

A persistent object can live after the program which created it has stopped. Persistent objects can even outlive different versions of the creating program, can outlive the disk system, the operating system, or even the hardware on which the OS was running when they were created.callback methods might need to create persistent C++ objects, that is, objects that continue to exist after the method exits. For example, a callback method might need to access an object created during a previous invocation. Or one callback method might need to access an object created by another callback method. To create persistent C++ objects in your S-function:

Create a pointer work vector to hold pointers to the persistent object between method invocations: