RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL
New Scheme Based On AICTE Flexible Curricula
Computer Science and Engineering, III-Semester

CS304 OBJECT ORIENTED PROGRAMMING SYSTEMS

1. Introduction to Object Oriented Thinking & Object Oriented Programming: Comparison with Procedural Programming, features of Object oriented paradigm– Merits and demerits of OO methodology; Object model; Elements of OOPS, IO processing.

2. Encapsulation and Data Abstraction- Concept of Objects: State, Behavior &

Identity of an object; Classes: identifying classes and candidates for Classes Attributes and Services, Access modifiers, Static members of a Class, Instances, Message passing, and Construction and destruction of Objects.

3. Relationships – Inheritance: purpose and its types, 'is a' relationship; Association, Aggregation. Concept of interfaces and Abstract classes.

4. Polymorphism: Introduction, Method Overriding & Overloading, static and run time Polymorphism.

5. Strings, Exceptional handling, Introduction of Multi-threading and Data collections. Case study like: ATM, Library management system.

Text Books

1. Timothy Budd, "An Introduction to Object-Oriented Programming", Addison-Wesley Publication, 3rd Edition.

2. Cay S. Horstmann and Gary Cornell, "Core Java: Volume I, Fundamentals", Prentice Hall publication.

Reference Books

1. G. Booch, "Object Oriented Analysis& Design", Addison Wesley.
2. James Martin, "Principles of Object Oriented Analysis and Design", Prentice Hall/PTR.
3. Peter Coad and Edward Yourdon, "Object Oriented Design", Prentice Hall/PTR.
4. Herbert Schildt, "Java 2: The Complete Reference", McGraw-Hill Osborne Media, 7th Ed.

Class Notes

UNIT I

1. Object Oriented Thinking:-

Traditionally, a programming problem is attacked by coming up with some kinds of data representations, and procedures that operate on that data. Under this model, data is inert, passive, and helpless; it sits at the complete mercy of a large procedural body, which is active, logical, and all-powerful.

The problem with this approach is that programs are written by programmers, who are only human and can only keep so much detail clear in their heads at any one time. As a project gets larger, its procedural core grows to the point where it is difficult to remember how the whole thing works. Minor lapses of thinking and typographical errors become more likely to result in well-concealed bugs. Complex and unintended interactions begin to emerge within the procedural core, and maintaining it becomes like trying to carry around an angry squid without letting any tentacles touch your face. There are guidelines for programming that can help to minimize and localize bugs within this traditional paradigm, but there is a better solution that involves fundamentally changing the way we work.

1.1 What is abstract interaction?

If you want to change the television channel from your seat, you use a remote control. That remote control is an **object** with a number of **attributes** and **behaviors** hidden inside of it. Without an understanding of those hidden attributes—the microchips, wiring, etc.—you still know and expect that pressing a button will perform that particular function. You've interacted with the remote control in the abstract, skipping the steps the remote was designed to carry out. That's the beauty of OOP—the focus is on how the objects behave, not the code required to tell them how to behave.

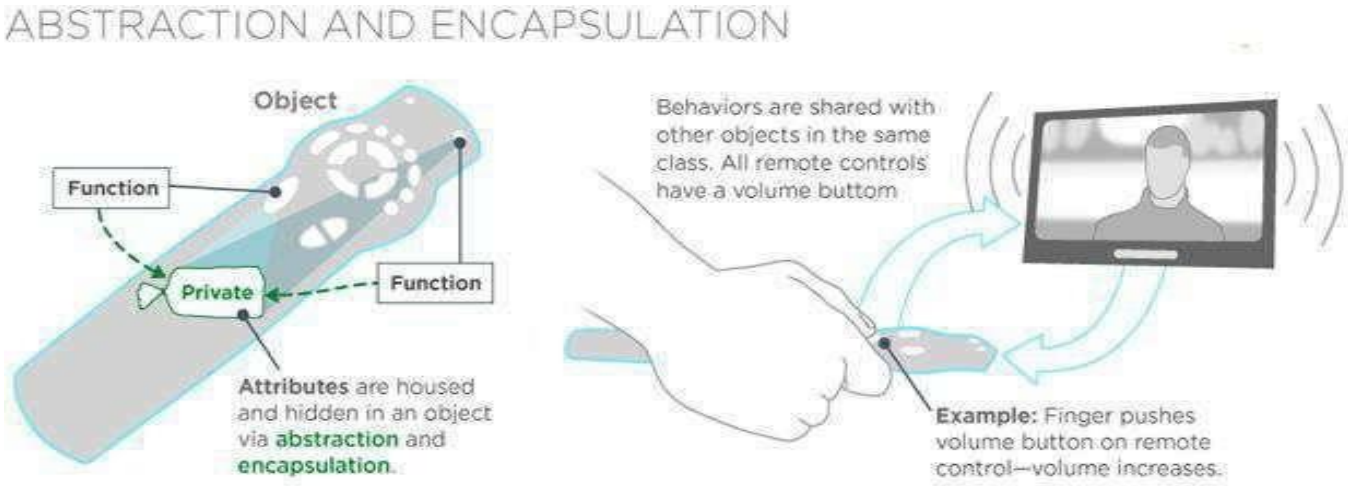


Figure1. Abstract interaction

2. What Are Objects?

A car is an example of a complex object, with many attributes. We don't need to understand all of its internal mechanics, what kind of engine it has, how the gas makes it run, or even where the gas came from in order to know how to interact with it. The car's behaviors have been made simple for us through object-oriented logic: put the key in the ignition, and the car turns on and gets us where we need to go. The attributes that make this possible—all of the car's parts, electronics, and engineering—are a "package" we don't need to break down in order to understand.

Apply this to software building, and it allows developers to break down big, complicated projects into compartmentalized objects, program them to have attributes and behaviors, then essentially set them aside and focus on programming how the objects interact—a higher level of thinking that makes writing code less linear and more efficient. Modern, high-level languages like Python and Ruby are perfect examples of OOP. The fact that they're able to be so streamlined gets right to the heart of OOP logic.

2 .Object-Oriented Programming & Back-End Development

What is object-oriented programming in terms of how a site is built? OOP defines most modern server-side scripting languages, which are the languages back-end developers use to write software and database technology. This behind-the-scenes, server-side technology tells a website or web application how to behave, and also builds the architecture for a site to interact with its database. That scaffolding is how data is delivered and processed, effectively making it the brain of a website. And that's where object-oriented logic comes into play.

If a website's brain uses object-oriented logic, it's designed to think of data as objects**.** It affects how a site is built from the ground up, how data is organized, how later growth and maintenance of the site will occur, and more.

2.1 Benefits of object-oriented technology include:

- Ease of software design
- Productivity
- Easy testing, debugging, and maintenance
- It's reusable
- More thorough data analysis, less development time, and more accurate coding, thanks to OOP's inheritance method
- Data is safe and secure, with less data corruption, thanks to hiding and abstraction.
- It's sharable (classes are reusable and can be distributed to other networks).

3. Difference Between Procedure Oriented Programming (Pop) &Object-Oriented Programming (Oop)

Object Oriented Programming	Procedure Oriented Programming	Points
In OOP, program is divided into parts called objects.	In POP, program is divided into small parts called functions.	Divided Into

In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.	In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	Importance
OOP follows Bottom Up approach.	POP follows Top Down approach.	Approach
OOP has access specifies named Public, Private, Protected, etc.	POP does not have any access specified.	Access Specifies
In OOP, objects can move and communicate with each other through member functions.	In POP, Data can move freely from function to function in the system.	Data Moving
OOP provides an easy way to add new data and function.	To add new data and function in POP is not so easy.	Expansion
In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	Data Access
OOP provides Data Hiding so provides more security.	POP does not have any proper way for hiding data so it is less secure.	Data Hiding
In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.	In POP, Overloading is not possible.	Overloading
Example of OOP is: C++, JAVA, VB.NET, C#.NET.	Example of POP is: C, VB, FORTRAN, and Pascal.	Examples

4 Principles Of Object-Oriented Systems

The conceptual framework of object–oriented systems is based upon the object model. There are two categories of elements in an object-oriented system −

Major Elements − By major, it is meant that if a model does not have any one of these elements, it ceases to be object oriented. The four major elements are −

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Minor Elements − By minor, it is meant that these elements are useful, but not indispensable part of the object model. The three minor elements are −

- Typing
- Concurrency
- Persistence

1) Abstraction

Abstraction means to focus on the essential features of an element or object in OOP, ignoring its extraneous or accidental properties. The essential features are relative to the context in which the object is being used

.

Grady Booch has defined abstraction as follows −

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

Example − When a class Student is designed, the attributes enrolment number, name, course, and address are included while characteristics like pulse_rate and size_of_shoe are eliminated, since they are irrelevant in the perspective of the educational institution.

2) Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. The class has methods that provide user interfaces by which the services provided by the class may be used.

3) Modularity

Modularity is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem. Booch has defined modularity as −

"Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."

Modularity is intrinsically linked with encapsulation. Modularity can be visualized as a way of mapping encapsulated abstractions into real, physical modules having high cohesion within the modules and their inter–module interaction or coupling is low.

4) Hierarchy

In Grady Booch's words, "Hierarchy is the ranking or ordering of abstraction". Through hierarchy, a system can be made up of interrelated subsystems, which can have their own subsystems and so on until the smallest level components are reached. It uses the principle of "divide and conquer". Hierarchy allows code reusability.

The two types of hierarchies in OOA are −

- **"IS–A" hierarchy** − It defines the hierarchical relationship in inheritance, whereby from a super-class, a number of subclasses may be derived which may again have subclasses and so on. For example, if we derive a class Rose from a class Flower, we can say that a rose "is–a" flower.
- **"PART–OF" hierarchy** − It defines the hierarchical relationship in aggregation by which a class may be composed of other classes. For example, a flower is composed of sepals, petals, stamens, and carpel. It can be said that a petal is a "part–of" flower.

5) Typing

According to the theories of abstract data type, a type is a characterization of a set of elements. In OOP, a class is visualized as a type having properties distinct from any other types. Typing is the enforcement of the notion that an object is an instance of a single class or type. It also enforces that objects of different types may not be generally interchanged; and can be interchanged only in a very restricted manner if absolutely required to do so.

The two types of typing are −

- **Strong Typing** − Here, the operation on an object is checked at the time of compilation, as in the programming language Eiffel.
- **Weak Typing** − Here, messages may be sent to any class. The operation is checked only at the time of execution, as in the programming language Smalltalk.

6) Concurrency

Concurrency in operating systems allows performing multiple tasks or processes simultaneously. When a single process exists in a system, it is said that there is a single thread of control. However, most systems have multiple threads, some active, some waiting for CPU, some suspended, and some terminated. Systems with multiple CPUs inherently permit concurrent threads of control; but systems running on a single CPU use appropriate algorithms to give equitable CPU time to the threads so as to enable concurrency.

In an object-oriented environment, there are active and inactive objects. The active objects have independent threads of control that can execute concurrently with threads of other objects. The active objects synchronize with one another as well as with purely sequential objects.

7) Persistence

An object occupies a memory space and exists for a particular period of time. In traditional programming, the lifespan of an object was typically the lifespan of the execution of the program that created it. In files or databases, the object lifespan is longer than the duration of the process creating the object. This property by which an object continues to exist even after its creator ceases to exist is known as persistence.

5 Structured Analysis vs. Object Oriented Analysis

The Structured Analysis/Structured Design (SASD) approach is the traditional approach of software development based upon the waterfall model. The phases of development of a system using SASD are −

- Feasibility Study
- Requirement Analysis and Specification
- System Design
- Implementation
- Post-implementation Review

Now, we will look at the relative advantages and disadvantages of structured analysis approach and object-oriented analysis approach.

5.1 Advantages/Disadvantages of Object Oriented Analysis

Advantages	Disadvantages
Focuses on data rather than the procedures as in Structured Analysis.	Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be	It cannot identify which objects would generate an optimal system design.

tampered by other parts of the system.	
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	The object-oriented models do not easily show the communications between the objects in the system.
It allows effective management of software complexity by the virtue of modularity.	All the interfaces between the objects cannot be represented in a single diagram.
It can be upgraded from small to large systems at a greater ease than in systems following structured analysis.	

5.2 Advantages/Disadvantages of Structured Analysis

Advantages	**Disadvantages**
As it follows a top-down approach in contrast to bottom-up approach of object-oriented analysis, it can be more easily comprehended than OOA.	In traditional structured analysis models, one phase should be completed before the next phase. This poses a problem in design, particularly if errors crop up or requirements change.
It is based upon functionality. The overall purpose is identified and then functional decomposition is done for developing the software. The emphasis not only gives a better understanding of the system but also generates more complete systems.	The initial cost of constructing the system is high, since the whole system needs to be designed at once leaving very little option to add functionality later.
The specifications in it are written in simple English language, and hence can be more easily analyzed by non-technical personnel.	It does not support reusability of code. So, the time and cost of development is inherently high.

6 . Advantages And Disadvantages Of Object-Oriented Programming (Oop)

This reading discusses advantages and disadvantages of object-oriented programming, which is a well-adopted programming style that uses interacting objects to model and solve complex programming tasks. Two examples of popular object-oriented programming languages are Java and C++. Some other well-known object-oriented programming languages include Objective C, Perl, Python, JavaScript, Simula, Modula, Ada, Smalltalk, and the Common Lisp Object Standard.

6.1 Some of the advantages of object-oriented programming include:

1. Improved software-development productivity: Object-oriented programming is modular, as it provides separation of duties in object-based program development. It is also extensible, as objects can be extended to include new attributes and behaviors. Objects can also be reused within an across

applications. Because of these three factors – modularity, extensibility, and reusability – object-oriented programming provides improved software-development productivity over traditional procedure-based programming techniques.

2. Improved software maintainability: For the reasons mentioned above, object- oriented software is also easier to maintain. Since the design is modular, part ofthe system can be updated in case of issues without a need to make large-scale changes.

3. Faster development: Reuse enables faster development. Object-oriented programming languages come with rich libraries of objects, and code developed during projects is also reusable in future projects.

4. Lower cost of development: The reuse of software also lowers the cost of development. Typically, more effort is put into the object-oriented analysis and design, which lowers the overall cost of development.

5. Higher-quality software: Faster development of software and lower cost of development allows more time and resources to be used in the verification of the software. Although quality is dependent upon the experience of the teams, object- oriented programming tends to result in higher-quality software.

6.2 Some of the disadvantages of object-oriented programming include:

1. Steep learning curve: The thought process involved in object-oriented programming may not be natural for some people, and it can take time to get used to it. It is complex to create programs based on interaction of objects. Some of the key programming techniques, such as inheritance and polymorphism, can be challenging to comprehend initially.

2. Larger program size: Object-oriented programs typically involve more lines of code than procedural programs.
3. Slower programs: Object-oriented programs are typically slower than procedure- based programs, as they typically require more instructions to be executed.

4. Not suitable for all types of problems: There are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style, and applying object-oriented programming in those situations will not result in efficient programs.

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see the whole world in the form of objects. For example, a car is an object which has certain properties such as color, the number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

7 . Characteristics Of Object-Oriented Programming:

1) **Object**
   This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as an object.

2) **Class**

When you define a class, you define a blueprint for an object. This doesn't define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

3) **Abstraction**
   Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.
   For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provide different methods to the outside world without giving internal detail about those methods and data.

4) **Encapsulation**
   Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you a framework to place the data and the relevant functions together in the same object.
5) **Inheritance**
   One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as a base class, a new class is formed called as derived class.
   This is a very important concept of object-oriented programming since this feature helps to reduce the code size.
6) **Polymorphism**
   The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.
7) **Overloading**
   The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

8 Object Model

An object model is a logical interface, software or system that is modeled through the use of object-oriented techniques. It enables the creation of an architectural software or system model prior to development or programming.

An object model is part of the object-oriented programming (OOP) lifecycle.

An object model helps describe or define a software/system in terms of objects and classes. It defines the interfaces or interactions between different models, inheritance, encapsulation and other object-oriented interfaces and features.

8.1 Object model examples include:

- **Document Object Model (DOM):** A set of objects that provides a modeled representation of dynamic HTML and XHTML-based Web pages

- **Component Object Model (COM)**: A proprietary Microsoft software architecture used to create software components

9 Primitive Built-In Types

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types:

Keyword	Type
Bool	Boolean
Char	Character
Int	Integer
Float	Floating point
Double	Double floating point
Void	Valueless
wchar_t	Wide character

9.1 Types of Modifier:-

Several of the basic types can be modified using one or more of these type modifiers:

- signed
- unsigned
- short
- long

9.2 Data type and with range:-

The following table shows the variable type, how much memory it takes to store the value in memory and what is a maximum and minimum value which can be stored in such type of variables.

Typical Range	Typical Bit Width	Type
-128 to 127 or 0 to 255	1byte	char
0 to 255	1byte	unsigned char
-128 to 127	1byte	signed char
-2147483648 to 2147483647	4bytes	int
0 to 4294967295	4bytes	unsigned int

-2147483648 to 2147483647	4bytes	signed int
-32768 to 32767	2bytes	short int
0 to 65,535	2bytes	unsigned short int
-32768 to 32767	2bytes	signed short int
-2,147,483,648 to 2,147,483,647	8bytes	long int
-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8bytes	signed long int
0 to 18,446,744,073,709,551,615	8bytes	unsigned long int
+/- 3.4e +/- 38 (~7 digits)	4bytes	float
+/- 1.7e +/- 308 (~15 digits)	8bytes	double
+/- 1.7e +/- 308 (~15 digits)	8bytes	long double
1 wide character	2 or 4 bytes	wchar_t

The sizes of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

9.3 Variable Declaration in C++

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable definition at the time of linking of the program.

Example

Try the following example where a variable has been declared at the top, but it has been defined inside the main function:

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

// Variable declaration:
externint a, b;
externint c;
extern float f;

int main () {
  // Variable definition:
```

```
int a, b;
int c;
float f;

   // actual initialization
   a = 10;
   b = 20;
   c = a + b;

cout<< c <<endl ;

   f = 70.0/3.0;
cout<< f <<endl ;

return 0;
}
```

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

```
30
23.3333
```

We will learn what a function is and it's parameter in subsequent chapters. Here let us explain what local and global variables are.

1. **Local Variables**

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables:

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

int main () {
   // Local variable declaration:
int a, b;
int c;

   // actual initialization
   a = 10;
   b = 20;
   c = a + b;

cout<< c;
```

```
return 0;
}
Output
30
```

2. Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.
A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

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

// Global variable declaration:
int g;

int main () {
  // Local variable declaration:
int a, b;

  // actual initialization
  a = 10;
  b = 20;
  g = a + b;

cout<< g;

return 0;
}
Output
30
```


10 C++ Arrays

In programming, one of the frequently arising problem is to handle numerous data of same type.
Consider this situation, you are taking a survey of 100 people and you have to store their age. To solve this problem in C++, you can create an integer array having 100 elements.
An array is a collection of data that holds fixed number of values of same type. For example:

```
int age[100];
```

Here, the *age* array can hold maximum of 100 elements of integer type.
The size and type of arrays cannot be changed after its declaration.

DataTypearrayName [arraySize];

float mark[5];

C++ program to store and calculate the sum of 5 numbers entered by the user using arrays.

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

int main()
{
   int numbers[5], sum = 0;
cout<< "Enter 5 numbers: ";

   //  Storing 5 number entered by user in an array
   //  Finding the sum of numbers entered
   for (int i = 0; i< 5; ++i)
   {
cin>> numbers[i];
       sum += numbers[i];
   }

cout<< "Sum = " << sum <<endl;

   return 0;
}
```


11 C++ Structures

Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collection of data of different data types.

For example: You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables *name, citNo, salary* to store these information separately.
However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: *name1, citNo1, salary1, name2, citNo2, salary2*
You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.
The struct keyword defines a structure type followed by an identifier (name of the structure).

struct Person

```
{
   char name[50];
   int age;
   float salary;
};
```

C++ Program to assign data to members of a structure variable and display it.

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

struct Person
{
   char name[50];
   int age;
   float salary;
};

int main()
{
   Person p1;

cout<< "Enter Full name: ";
cin.get(p1.name, 50);
cout<< "Enter age: ";
cin>> p1.age;
cout<< "Enter salary: ";
cin>> p1.salary;

cout<< "\nDisplaying Information." <<endl;
cout<< "Name: " << p1.name <<endl;
cout<<"Age: " << p1.age <<endl;
cout<< "Salary: " << p1.salary;

   return 0;
}
```


……End of Unit 1……..

Encapsulation and Data Abstraction:

Definition: It is the method of combining the data and functions inside a class. Thus the data gets hidden from being accessed directly from outside the class. This is similar to a capsule where several medicines are kept inside the capsule thus hiding them from being directly consumed from outside. All the members of a class are private by default, thus preventing them from being accessed from outside the class.

Why Encapsulation

Encapsulation is necessary to keep the details about an object hidden from the users of that object. Details of an object are stored in its data members. This is the reason we make all the member variables of a class private and most of the member functions public. Member variables are made private so that these cannot be directly accessed from outside the class and so most member functions are made public to allow the users to access the data members through those functions.

For example, we operate a washing machine through its power button. We switch on the power button, the machine starts and when we switch it off, the machine stops. We don't know what mechanism is going on inside it. That is encapsulation.

Benefits of Encapsulation

There are various benefits of **encapsulated classes**.

- Encapsulated classes reduce complexity.
- **Help protect our data.** A client cannot change an Account's balance if we encapsulate it.
- **Encapsulated classes are easier to change.** We can change the privacy of the data according to the requirement without changing the whole program by using access modifiers (public, private, protected). For example, if a data member is declared private and we wish to make it directly accessible from anywhere outside the class, we just need to replace the specifier private by public.

Let's see an example of Encapsulation.

```
#include<iostream>

usingnamespacestd;

classRectangle
{
        intlength;
        intbreadth;
        public:
                voidsetDimension(intl,intb)
                {
                        length=l;
                        breadth=b;
                }
```

```
                intgetArea()
                {
                        returnlength*breadth;
                }
};

intmain()
{
        Rectanglert;
        rt.setDimension(7,4);
        cout<<rt.getArea()<<endl;
        return0;
}
```

Data Abstraction

Data abstraction is one of the most essential and important feature 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 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. .

Advantages of Data Abstraction:

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

Concept of Objects: State, Behavior &Identity of an object

Object: Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality. Object is a runtime entity, it is created at runtime. Object is an instance of a class. All the members of the class can be accessed through object.

Bundling code into individual software objects provides a number of benefits, including:

1. **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. **Information-hiding:** By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. **Code re-use:** If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. **Pluggability and debugging ease:** If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its

replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

State: Represents data (value) of an object. Dogs have state (name, color, and breed, hungry)

Behavior: Represents the behavior (functionality) of an object such as deposit, withdraw etc.

Identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But it is used internally by the JVM to identify each object uniquely.

Classes: Identifying classes and candidate for classes Attribute and services

Class: The building block of C++ that leads to Object Oriented programming is a **Class**.

- A class is an abstract data type similar to '**C structure**'.
- The Class representation of objects and the sets of operations that can be applied to such objects.
- The class consists of Data members and methods.

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.

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.

- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage etc and member functions can be apply brakes, increase speed etc.

The primary purpose of a class is to hold data/information. This is achieved with attributes which are also known as data members. The member functions determine the behavior of the class i.e. provide a definition for supporting various operations on data held in form of an object.

Class Advantages

- A class is a user-defined and reusable entity.
- A class reduces complicity of holding data.
- Classes' just holds data and do operations with help of member variables and member functions.
- A class maintains software quality and is well structured.
- An Object is a realization of the particular class.

Syntax

classclassname {

```
   Access - Specifier :
   Member Varibale Declaration;
   Member Function Declaration;
}
```

Example

```
#include<iostream>
usingnamespacestd;

// Class Declaration

class Person {
//Access - Specifier
public:
//Member Variable Declaration
string name;
int number;

//Member Functions read() and print() Declaration

voidread(){
//Get Input Values For Object Variables
cout<<"Enter the Name :";
cin>> name;

cout<<"Enter the Number :";
cin>> number;
  }

voidprint(){
//Show the Output
cout<<"Name : "<< name <<",Number : "<< number <<endl;
  }
};

intmain(){
// Object Creation For Class
  Person obj;

cout<<"Simple Class and Object Example Program In C++\n";

obj.read();
obj.print();

return0;
}
```


Access Modifiers

Access modifiers are used to implement important feature of Object Oriented Programming known as **Data Hiding**. Consider a real life example: What happens when a driver applies

brakes? The car stops. The driver only knows that to stop the car, he needs to apply the brakes. He is unaware of how actually the car stops. That is how the engine stops working or the internal implementation on the engine side. This is what data hiding is.
Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.
There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

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.
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.
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.

Demonstrating use of private and public Data members and Member functions


```
#include <iostream>
usingnamespace std;

class Example
{
        private:
                intval;
        public:
                //function declarations
                voidinit_val(int v);
                voidprint_val();
};
//function definitions
void Example::init_val(int v)
{
        val=v;
}

void Example::print_val()
{
        cout<<"val: "<<val<<endl;
}
int main()
{
        //create object
```

```
        Example Ex;
        Ex.init_val(100);
        Ex.print_val();

        return 0;
}
```

Static Members of Class: Data member and Member function

A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are:

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects. Program illustrates the use of a static data member.

```
#include
using namespace std;

class item
{
        staticint count;
        int number;
        public:
                voidgetdata(int a)
                {
                        number = a;
                        count ++;
                }
                voidgetcount(void)
                {
                        cout<< "Count: ";
                        cout<< count <<"\n";
                }
};
int item :: count;

int main()
{
        itema,b,c;
        a.getcount();
        b.getcount();
        c.getcount();

        a.getdata(100);
        b.getdata(200);
```

```
        c.getdata(300);

        cout<< "After reading data"<<"\n";
        a.getcount();
        b.getcount();
        c.getcount();
        return 0;
}
```

The output of the program would be:
Count: 0
Count: 0
Count: 0
After reading data
Count: 3
Count: 3
Count: 3

Static Member Function

Like static member variable, we can also have static member function. A member function that is declared static has the following properties:

- A static function can have access to only static members declared in the class.
- A static member function can be called the class name as follows:

Class-name:: function-name

Static member functions are used to maintain a single copy of a class member function across various objects of the class. Static member functions can be called either by itself, independent of any object, by using class name and :: (scope resolution operator) or in connection with an object.

Characteristic static member functions are:

- A static member function can only have access to other static data members and functions declared in the same class.
- A static member function can be called using the class name with a scope resolution operator instead of object name.
- Global functions and data may be accessed by static member function.
- A static member function does not have this Pointer.
- There can not be a static and a non-static version of the same function.
- They cannot be declared as const or volatile.
- A static member function may not be virtual.

Instance

An instance is an object of a class type created via the new operator.
For example:

String *str = new String();
str is a pointer to an instance of class String.
if you won`t create memory for an object so we call that **object** as **instance**.
example: Student std;
here**Student** is a class and **std** is a instance(means a just a copy that class),with this we won`t do anything until unless we create a memory for that.
s

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.

Message Passing is nothing but sending and receiving of information by the objects same as people exchange information. So this helps in building systems that simulate real life. Following are the basic steps in message passing.

- Creating classes that define objects and its behavior.
- Creating objects from class definitions
- Establishing communication among objects

In OOPs, Message Passing involves specifying the name of objects, the name of the function, and the information to be sent.

Construction and Destruction of Object

The process of initializing and starting up a class is called construction. The opposite of this is the process of tearing down a class and cleaning it up, which is called destruction or finalization. Class construction occurs either when an object is being initialized or upon first reference to a type.

In typical case, the process is as follows:

- Calculate the size of an object - the size is mostly the same as that of the class but can vary. When the object in question is not derived from a class, but from a prototype instead, the size of an object is usually that of the internal data structure (a hash for instance) that holds its slots.
- allocation - allocating memory space with the size of an object plus the growth later, if possible to know in advance
- binding methods - this is usually either left to the class of the object, or is resolved at dispatch time, but nevertheless it is possible that some object models bind methods at creation time.
- calling an initializing code of super class
- calling an initializing code of class being created

Construction Rules

The rule for constructing an object of a simple class is:
1. Call the constructor/initializer for each data member, in sequence.

2. Call the constructor for the class.

The rule for constructing an object of a derived class is:
1. Call the constructor for the base class (which recursively calls the constructors needed to Completely initialize the base class object.)
2. Call the constructor/initializer for each data member of the derived class, in sequence.
3. Call the constructor for the derived class

Object Destruction

It is generally the case that after an object is not used, it is removed from memory to make room for other programs or objects to take that object's place. However, if there is sufficient memory or a program has a short run time, object destruction may not occur, memory simply being deallocated at process termination. In some cases object destruction simply consists of deallocating the memory, particularly in garbage-collected languages, or if the "object" is actually a plain old data structure. In other cases some work is performed prior to deallocation, particularly destroying member objects (in manual memory management), or deleting references from the object to other objects to decrement reference counts (in reference counting). This may be automatic, or a special destruction method may be called on the object.

Destruction Rules

When an object is deleted, the destructors are called in the opposite order.
The rule for an object of a derived class is:
1. Call the destructor for the derived class.
2. Call the destructor for each data member object of the derived class in reverse sequence.
3. Call the destructor for the base class.

Constructor

A constructor is a special member function of the class which has the same name as that of the class. It is automatically invoked when we declare/create new objects of the class. It is called constructor, because it constructs the value of data members of the class.

The constructor functions have some special characteristics as follows –

1. They should be declared in the public section.
2. They are called automatically when the objects are created.
3. They do not have return types, not even void and they cannot return values.
4. They cannot be inherited, though a derived class can call the base class constructor.
5. Like other C++ functions, they can have default arguments.
6. Constructors cannot be virtual.
7. They cannot be referred by their addresses.
8. They make ‘implicit calls’ to the operators new and delete when memory allocation is required.

Syntax

classclass_name {

```
public:
class_name() {
        // Constructor code
    }

    //... other Variables & Functions
}
```

Types Of Constructor

- Default Constructor
- Parameterized Constructor
- Copy Constructor

Default Constructor

A default constructor does not have any parameters or if it has parameters, all the parameters have default values.

Syntax

```
classclass_name {
public:
    // Default constructor with no arguments
class_name();
    // Other Members Declaration
}
```

Parameterized Constructor

If a Constructor has parameters, it is called a Parameterized Constructor. Parameterized Constructors assist in initializing values when an object is created.

Example Program for Parameterized Constructor In C++

```
#include<iostream>
#include <iostream>
using namespace std;
class Employee {
public:
int id;//data member (also instance variable)
string name;//data member(also instance variable)
float salary;
Employee(int i, string n, float s)
        {
id = i;
name = n;
salary = s;
        }
void display()
        {
```

```
cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }
};
int main(void) {
   Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
   Employee e2=Employee(102, "Nakul", 59000);
e1.display();
e2.display();
return 0;
}
```

Copy Constructor

A copy constructor is a like a normal parameterized Constructor, but which parameter is the same class object. Copy constructor uses to initialize an object using another object of the same class.

The copy constructor is used to −

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

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

ClassName (constClassName&old_obj);

Following is a simple example of copy constructor.

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

class Point
{
private:
   int x, y;
public:
   Point(int x1, int y1) { x = x1; y = y1; }

   // Copy constructor
   Point(const Point &p2) {x = p2.x; y = p2.y; }

   intgetX()          {  return x; }
   intgetY()          {  return y; }
};

int main()
{
   Point p1(10, 15); // Normal constructor is called here
   Point p2 = p1; // Copy constructor is called here
```

```
    // Let us access values assigned by constructors
    cout<< "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout<< "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```

Destructor

Destructor: it is a member function which destructs or deletes an object. The destructor function has the same as the constructor, but it is preceded by a (tilde sign)

1. Destructors are special member functions of the class required to free the memory of the object whenever it goes out of scope.
2. Destructors are parameter less functions.
3. Name of the Destructor should be exactly same as that of name of the class. But preceded by '~' (tilde).
4. Destructor does not have any return type. Not even void.
5. The Destructor of class is automatically called when object goes out of scope.

Example

```
#include<iostream>
usingnamespace std;

class Marks
{
        public:
                intmaths;
                int science;

        //constructor
        Marks()
        {

                cout<< "Inside Constructor"<<endl;
                cout<< "C++ Object created"<<endl;
        }

  //Destructor
  ~Marks()
{
        cout<< "Inside Destructor"<<endl;
        cout<< "C++ Object destructed"<<endl;
  }
};

int main( )
{
  Marks m1;
  Marks m2;
return 0;
```

```
}
```

Output

```
Inside Constructor
C++ Object created
Inside Constructor
C++ Object created

Inside Destructor
C++ Object destructed
Inside Destructor
C++ Object destructed
```

Unit-III

Relationships: Inheritance- purpose and its types (IS-A Relationship)

One of the advantages of Object-Oriented programming language is code reuse. This reusability is possible due to the relationship b/w the classes. Object oriented programming generally support 4 types of relationships that are: inheritance, association, composition and aggregation. All these relationship is based on "is a" relationship, "has-a" relationship and "part-of" relationship.

Inheritance is one of the key features of Object-oriented programming in C++. It allows user to create a new class (derived class) from an existing class (base class).The derived class inherits all the features from the base class and can have additional features of its own.

The idea of inheritance implements the **IS-A** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

The capability of a class to derive properties and characteristics from another class is called **Inheritance**.

NOTE: All members of a class except Private, are inherited

Super Class: In the inheritance the class which is give data members and methods is known as base or super or parent class.

Sub Class: The class which is taking the data members and methods is known as sub or derived or child class.

Real Life Example of Inheritance in C++

The real life example of inheritance is child and parents, all the properties of father are inherited by his son.

Advantages of inheritance are as follows:

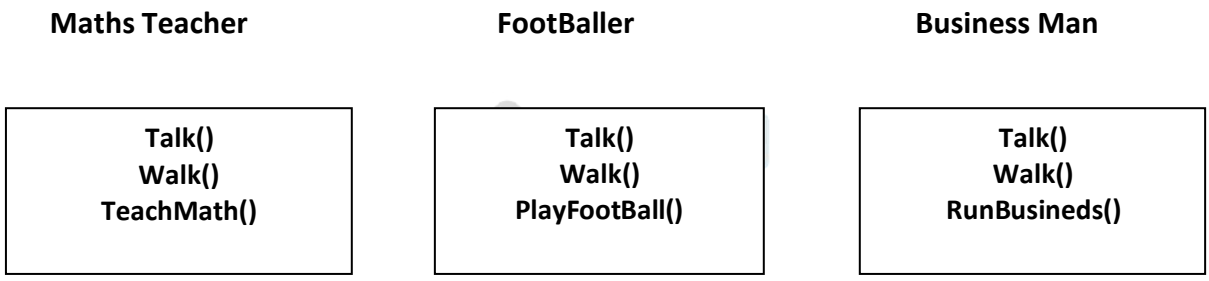
- Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of inherited class.
- Reusability enhanced reliability. The base class code will be already tested and debugged.
- As the existing code is reused, it leads to less development and maintenance costs.
- Inheritance makes the sub classes follow a standard interface.
- Inheritance helps to reduce code redundancy and supports code extensibility.
- Inheritance facilitates creation of class libraries.
- Application development time is less.
- Applications take less memory.
- Application execution time is less.
- Application performance is enhance (improved).
- Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.

Disadvantages of inheritance are as follows:

- Inherited functions work slower than normal function as there is indirection.
- Improper use of inheritance may lead to wrong solutions.
- Often, data members in the base class are left unused which may lead to memory wastage.
- Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes.
- One of the main disadvantages of inheritance in Java (the same in other object-oriented languages) is the increased time/effort it takes the program to jump through all the levels of overloaded classes. If a given class has ten levels of abstraction above it, then it will essentially take ten jumps to run through a function defined in each of those classes

Why inheritance should be used?

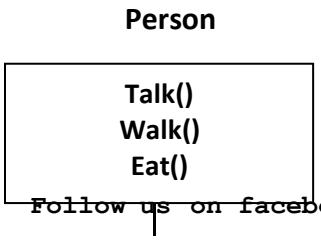
Suppose, in your game, you want three characters - a **maths teacher**, a **footballer** and a **businessman**.
Since, all of the characters are persons, they can walk and talk. However, they also have some special skills. A maths teacher can **teach maths**, a footballer can **play football** and a businessman can **run a business**.
You can individually create three classes who can walk, talk and perform their special skill as shown in the figure below.



In each of the classes, you would be copying the same code for walk and talk for each character.

If you want to add a new feature - eat, you need to implement the same code for each character. This can easily become error prone (when copying) and duplicate codes.

It'd be a lot easier if we had a **Person** class with basic features like talk, walk, eat, sleep, and add special skills to those features as per our characters. This is done using inheritance.



Using inheritance, now you don't implement the same code for walk and talk for each class. You just need to **inherit** them.
So, for Maths teacher (derived class), you inherit all features of a Person (base class) and add a new feature **TeachMaths**. Likewise, for a footballer, you inherit all the features of a Person and add a new feature **PlayFootball** and so on.
This makes your code cleaner, understandable and extendable.
It is important to remember: When working with inheritance, each derived class should satisfy the condition whether it **"is a"** base class or not. In the example above, Maths teacher **is a** Person, Footballer **is a** Person. You cannot have: Businessman **is a** Business.

The general form of defining a derived class is:

class **derived-class_name** : visivility-mode **base-class_name**
{
. . . . // members of the derived class
. . . .
};

A derived class inherits all base class methods with the following exceptions −

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

C++ offers five types of Inheritance. They are:

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied −

- **Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.

- **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Table showing all the Visibility Modes

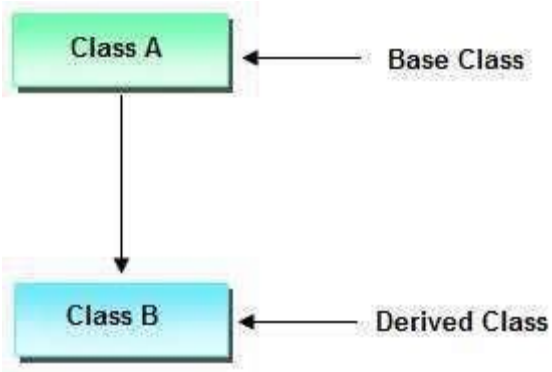
	Derived Class	**Derived Class**	**Derived Class**
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Type of Inheritance

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance (also known as Virtual Inheritance)

Single Inheritance

In single inheritance, there is only one base class and one derived class. The Derived class gets inherited from its base class. This is the simplest form of inheritance. The below figure is the diagram for single inheritance



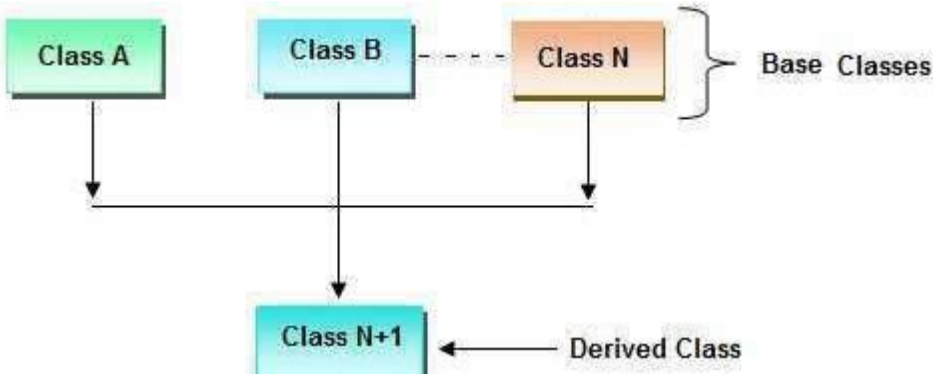
Program

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

```
class AddData          //Base Class
{
      protected:
            int num1, num2;
      public:
            void accept()
            {
                  cout<<"\n Enter First Number : ";
                  cin>>num1;
                  cout<<"\n Enter Second Number : ";
                  cin>>num2;
            }
};
class Addition: public AddData  //Class Addition – Derived Class
{
            int sum;
      public:
            void add()
            {
                  sum = num1 + num2;
            }
            void display()
            {
                  cout<<"\n Addition of Two Numbers : "<<sum;
            }
};
int main()
{
      Addition a;
      a.accept();
      a.add();
      a.display();
      return 0;
}
```


Multiple Inheritance

In this type of inheritance, a single derived class may inherit from two or more base classes. The below figures is the structure of Multiple Inheritance.



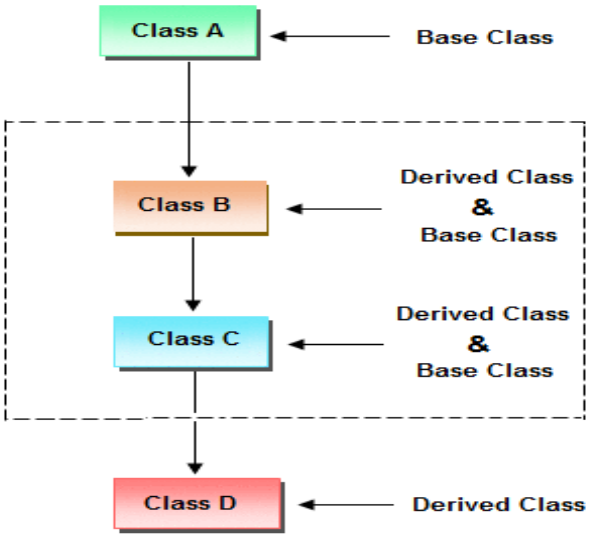
Program

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

class stud {
protected:
   int roll, m1, m2;

public:
   void get()
   {
      cout << "Enter the Roll No.: "; cin >> roll;
      cout << "Enter the two highest marks: "; cin >> m1 >> m2;
   }
};
class extracurriculam {
protected:
   int xm;

public:
   void getsm()
   {
      cout << "\nEnter the mark for Extra Curriculam Activities: "; cin >> xm;
   }
};
class output : public stud, public extracurriculam {
   int tot, avg;

public:
   void display()
   {
      tot = (m1 + m2 + xm);
      avg = tot / 3;
      cout << "\n\n\tRoll No    : " << roll << "\n\tTotal      : " << tot;
      cout << "\n\tAverage    : " << avg;
   }
};
```

```
int main()
{
   output O;
   O.get();
   O.getsm();
   O.display();
   return 0;
}
```

Multilevel Inheritance

The classes can also be derived from the classes that are already derived. This type of inheritance is called multilevel inheritance.

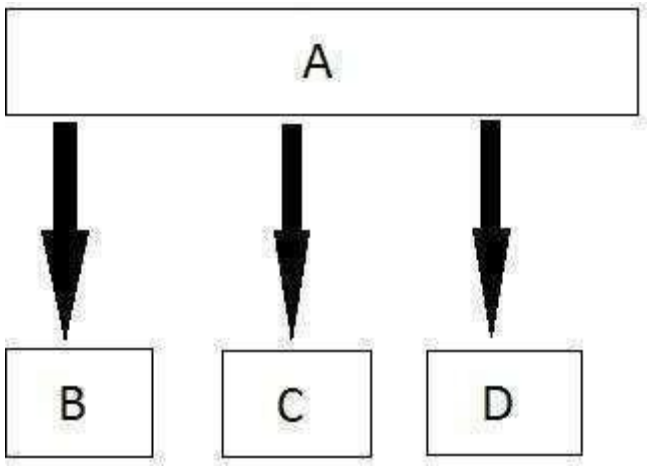



```
#include<iostream>
using namespace std;
class Student
{
        protected:
                int marks;
        public:
                void accept()
                {
                        cout<<" Enter marks";
                       cin>>marks;
                }
};
 class Test :public Student
{
         protected:
                int p=0;
        public:
                void check()
                {
```

```
                        if(marks>60)
                        {
                                p=1;
                        }
                }
};
class Result :public Test
{
        public:
                void print()
                {
                        if(p==1)
                                cout<<"\n You have passed";
                        else
                                cout<<"\n You have not passed";
                }
};
int main()
{
        Result r;
        r.accept();
        r.check();
        r.print();
        return 0;
}
```


Hierarchical Inheritance

In this type of inheritance, multiple derived classes get inherited from a single base class. The below fig is the structure of Hierarchical Inheritance.



Example

```
#include <iostream>
#include <string.h>
```

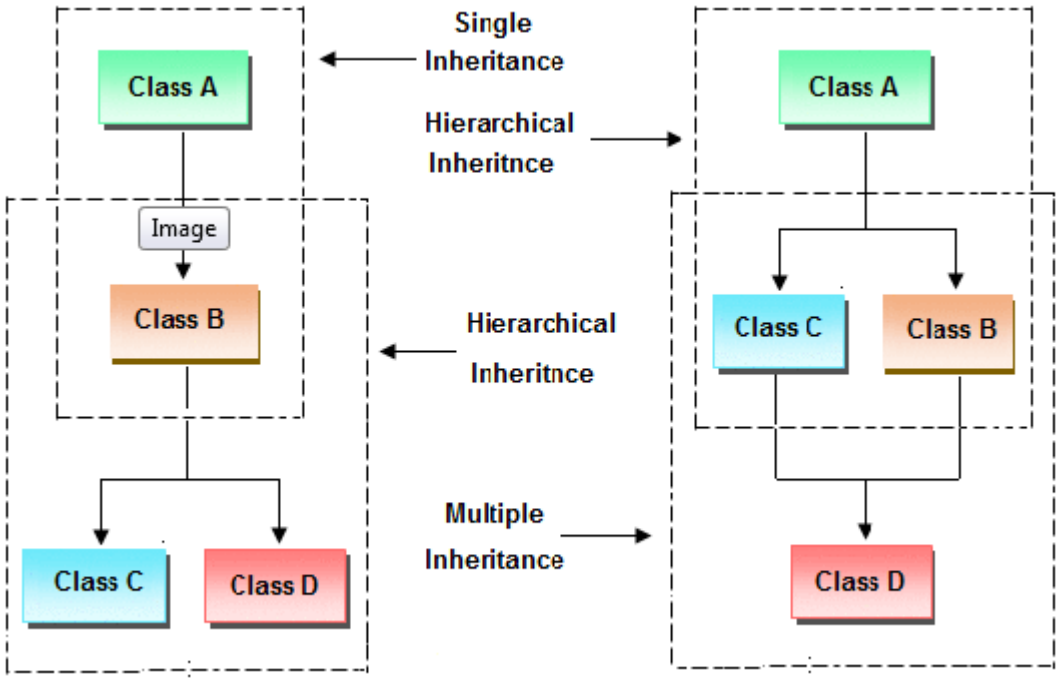
```
using namespace std;

class member {
   char gender[10];
   int age;

public:
   void get()
   {
      cout << "Age: "; cin >> age;
      cout << "Gender: "; cin >> gender;
   }
   void disp()
   {
      cout << "Age: " << age << endl;
      cout << "Gender: " << gender << endl;
   }
};
class stud : public member {
   char level[20];

public:
   void getdata()
   {
      member::get();
      cout << "Class: "; cin >> level;
   }
   void disp2()
   {
      member::disp();
      cout << "Level: " << level << endl;
   }
};
class staff : public member {
   float salary;

public:
   void getdata()
   {
      member::get();
      cout << "Salary: Rs."; cin >> salary;
   }
   void disp3()
   {
      member::disp();
      cout << "Salary: Rs." << salary << endl;
   }
};
int main()
{
   member M;
   staff S;
```

```
    stud s;
    cout << "Student" << endl;
    cout << "Enter data" << endl;
    s.getdata();
    cout << endl<< "Displaying data" << endl;
    s.disp();
    cout << endl<< "Staff Data" << endl;
    cout << "Enter data" << endl;
    S.getdata();
    cout << endl<< "Displaying data" << endl;
    S.disp();
}
```

Hybrid Inheritance

This is a Mixture of two or More Inheritance and in this Inheritance, a Code May Contains two or three types of inheritance in Single Code. In the below figure, the fig is the diagram for Hybrid inheritance.

It is a combination of two types of inheritance namely the multiple and hierarchical Inheritance. The following is a schematic representation.



Hybrid Inheritance is a method where one or more types of inheritance are combined together and used.

Program

```
#include<iostream>
```

```
Using namespace std;

class arithmetic
{
      protected:
           int num1, num2;
     public:
           void getdata()
          {
               cout<<"For Addition:";
               cout<<"\nEnter the first number: ";
               cin>>num1;
               cout<<"\nEnter the second number: ";
               cin>>num2;
         }
};
class plus:public arithmetic
{
          protected:
               int sum;
          public:
              void add()
             {
                 sum=num1+num2;
             }
};
class minus
{
            protected:
                 int n1,n2,diff;
            public:
                void sub()
                {
                    cout<<"\nFor Subtraction:";
                    cout<<"\nEnter the first number: ";
                    cin>>n1;
                    cout<<"\nEnter the second number: ";
                    cin>>n2;
                    diff=n1-n2;
                }
};
class result:public plus, public minus
{
              public:
                  void display()
```

```
        {
            cout<<"\nSum of "<<num1<<" and "<<num2<<"= "<<sum;
            cout<<"\nDifference of "<<n1<<" and "<<n2<<"= "<<diff;
        }
};
int main()
{
        result z;
        z.getdata();
        z.add();
        z.sub();
        z.display();
        return 0;
}
```

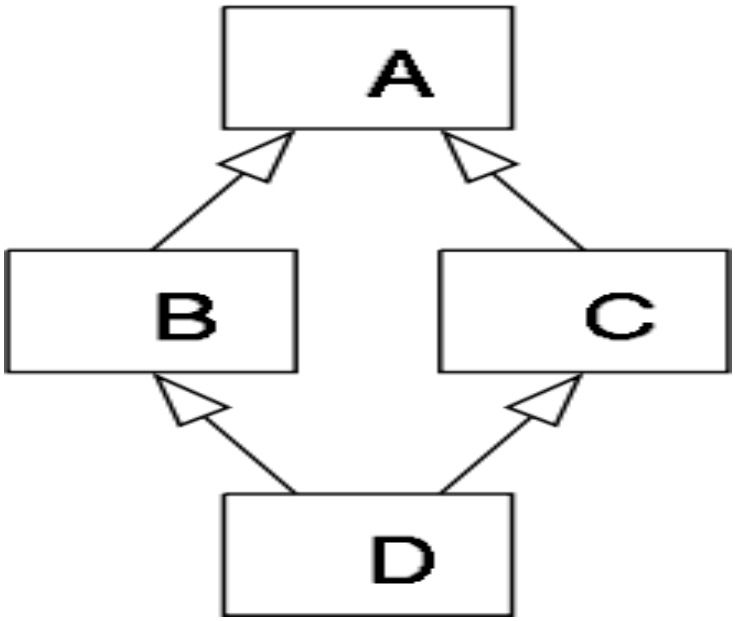
Diamond Problem in Inheritance

Let's understand this with one example.

```
class A {
   void display()
   {
      //some code
   }
}

class B : public A{
   void display()
   {
      //some code
   }
}

class C : public A{
   void display()
   {
      //some code
   }
}

class D : public B, public C{
   //contains two display() functions
}
```

Suppose there are four classes A, B, C and D. Class B and C inherit class A. Now class B and C contains one copy of all the functions and data members of class A.
Class D is derived from Class B and C. Now class D contains two copies of all the functions and data members of class A. One copy comes from class B and another copy comes from class C.
Let's say class A have a function with name display(). So class D have two display() functions as I have explained above. If we call display() function using class D object then ambiguity occurs

because compiler gets confused that whether it should call display() that came from class B or from class C. If you will compile above program then it will show error.



How to Remove Diamond Problem in C++?

We can remove diamond problem by using **virtual** keyword. It can be done in following way.


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

class A {
  public:
    void display()
    {
      cout<<"A Class Display Called.."<<endl;
    }
};

class B : virtual public A{
    public:
    void display()
    {
       cout<<"B Class Display Called.."<<endl;

    }
};

class C :virtual public A{
    public:

    void display()
    {
       cout<<"C Class Display Called.."<<endl;
```

```
    }
};

class D : public B, public C{
    //contains two display() functions
};

int main()
{
        A *a=new D();
        B *b=new D();
        C *c=new D();
        a->display();
        b->display();
        c->display();

        return 0;
}
```

Association

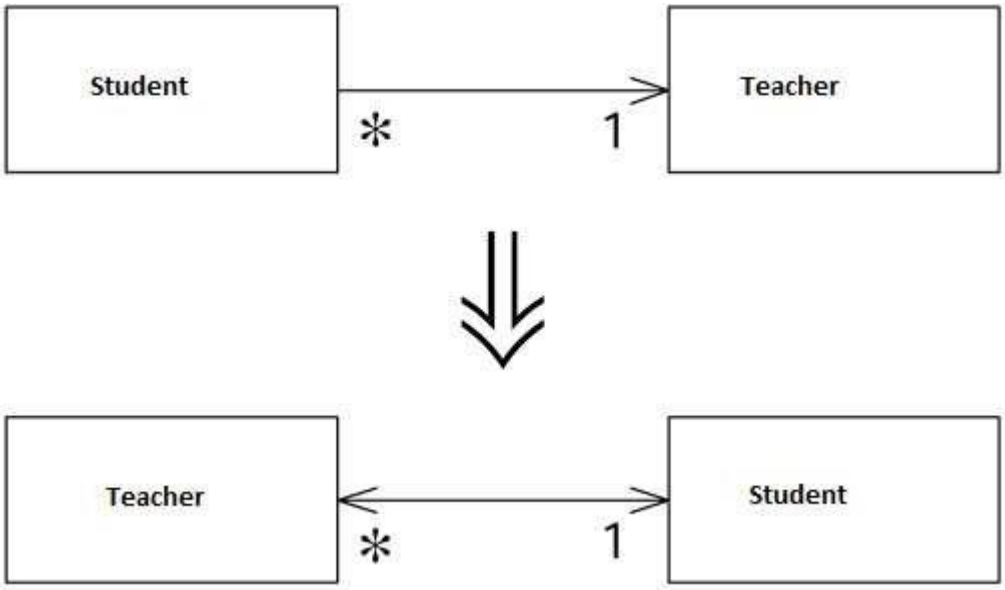
Association is a semantically weak relationship (a semantic dependency) between otherwise unrelated objects. An association is a "using" relationship between two or more objects in which the objects have their own life time and there is no owner. As an example, imagine the relationship between a doctor and a patient. A doctor can be associated with multiple patients and at the same time, one patient can visit multiple doctors for treatment and/or consultation. Each of these objects has their own life-cycle and there is no owner. In other words, the objects that are part of the association relationship can be created and destroyed independently.

In UML an association relationship is represented by a single arrow. An association relationship can be represented as (also known as cardinality) one-to-one, one-to-many and many-to-many. Essentially, an association relationship between two or more objects denotes a path of communication (also called a link) between them so that one object can send a message to another.

Association is a relationship which describes the reasons for the relationship and the rules that govern the relationship.

Let’s take an example of Teacher and Student.

Unidirectional Association is a specialized form of association where one object is associated with another object, but the reverse is not true. It is like one way communication.



Let’s take examples of multiple students can associate with a single teacher.

Bidirectional Association is a type of association where one object is related with other objects and the reverse is also true. It is like two way communication.

Let’s take examples of multiple students can associate with a single teacher and a single student can associate with multiple teachers.

But there is no ownership between the objects and both have their own life cycle. Both can create and delete independently.

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.

Let’s 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.

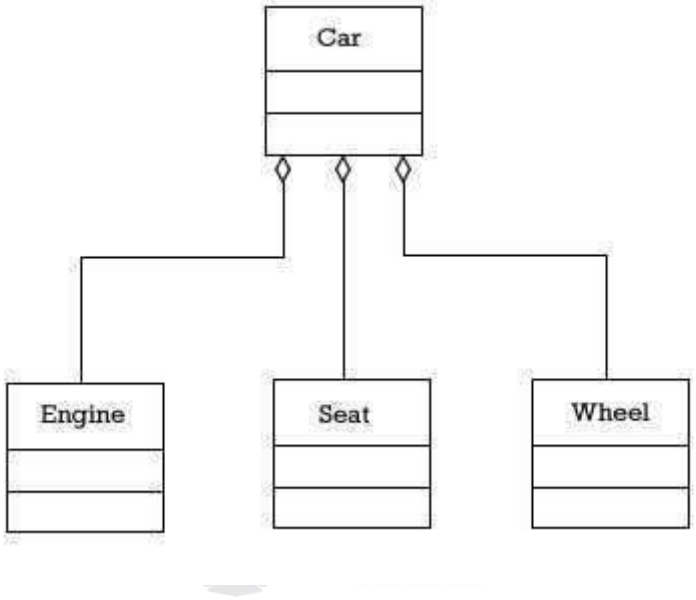
Aggregation (HAS-A **Relationship)**

Aggregation is a specialized form of association between two or more objects in which the objects have their own life-cycle but there exists an ownership as well. Aggregation is a typical whole/part relationship but it may or may not denote physical containment -- the objects may or may or be a part of the whole. In aggregation the objects have their own life-cycle but they do have ownership as well. As an example, an employee may belong to multiple departments in the organization. However, if the department is deleted, the employee object wouldn't be destroyed. Note that objects participating in an aggregation relationship cannot have cyclic aggregation relationships, i.e., a whole can contain a part but the reverse is not true.

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.

car object is an aggregation of engine, seat, wheels and other objects.



When do we use Aggregation??

Code reuse is best achieved by aggregation.

C++ Aggregation Implementation

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

```
 #include <iostream>
using namespace std;
class Address {
    public:
   string addressLine, city, state;
     Address(string addressLine, string city, string state)
    {
       this->addressLine = addressLine;
       this->city = city;
       this->state = state;
    }
};
class Employee
{
    private:
       Address* address;  //Employee HAS-A Address
```

```
    public:
    int id;
    string name;
    Employee(int id, string name, Address* address)
    {
       this->id = id;
       this->name = name;
       this->address = address;
    }
   void display()
    {
       cout<<id <<" "<<name<< " "<<
         address->addressLine<< " "<< address->city<< " "<<address->state<<endl;
    }
  };
int main(void) {
   Address a1= Address("C-146, Sec-15","Noida","UP");
   Employee e1 = Employee(101,"Nakul",&a1);
   e1.display();
    return 0;
}
```

Composition

Composition is a specialized form of aggregation in which if the parent object is destroyed, the child objects would cease to exist. It is actually a strong type of aggregation and is also referred to as a "death" relationship. As an example, a house is composed of one or more rooms. If the house is destroyed, all the rooms that are part of the house are also destroyed as they cannot exist by themselves.

A good real-life example of a composition is the relationship between a person's body and a heart. Let's examine these in more detail.

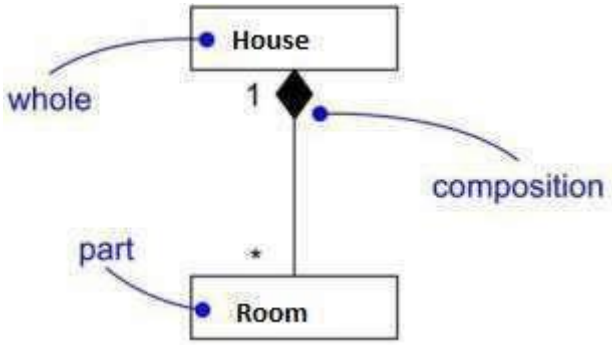
Composition relationships are part-whole relationships where the part must constitute part of the whole object. For example, a heart is a part of a person's body. The part in a composition can only be part of one object at a time. A heart that is part of one person's body cannot be part of someone else's body at the same time.

Object Composition

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc… Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on. This process of building complex objects from simpler ones is called **object composition**.

Broadly speaking, object composition models a "has-a" relationship between two objects. A car "has-a" transmission. Your computer "has-a" CPU. You "have-a" heart. The complex object is sometimes called the whole, or the parent. The simpler object is often called the part, child, or component.

Example "engine is part of car", "heart is part of body".



Implementation

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

using namespace std;

class Birthday{

public:
   Birthday(int cmonth, int cday, int cyear){
      cmonth = month;
      cday = day;
      cyear = year;

   }
   void printDate(){
      cout<<month <<"/" <<day <<"/" <<year <<endl;

   }
private:
   int month;
   int day;
   int year;

};

class People{

public:
   People(string cname, Birthday cdateOfBirth):name(cname), dateOfBirth(cdateOfBirth)
   {

   }
   void printInfo(){
      cout<<name <<" was born on: ";
      dateOfBirth.printDate();
   }
```

```
private:
   string name;
   Birthday dateOfBirth;

};


int main() {

   Birthday birthObject(7,9,97);
   People infoObject("Shantilal", birthObject);
   infoObject.printInfo();
   return 0;
}
```

Comparison Chart

	Association	**Aggregation**	**Composition**
Owner	No owner	Single owner	Single owner
Life time	Have their own lifetime	Have their own lifetime	Owner's life time
Child object	Child objects all are independent	Child objects belong to a single parent	Child objects belong to a single parent

Concept of interfaces and Abstract classes

To make it short, there is no 'interface' in C++. Conceptually, an interface is a pure virtual class, with no implementation whatsoever.

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. **Abstract class**
2. **Interface**

Abstract class and interface both can have abstract methods which are necessary for abstraction.

Abstract Class

An abstract class is a class that is designed to be specifically used as a base class. An abstract class is a class that has at least a pure virtual method. You can't create instances of that class, but you can have implementation in it, that is shared with the derived classes.

In C++ 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. Its implementation must be provided by derived classes.

Implementation

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

// Base class
class Shape {
  public:
    // pure virtual function providing interface framework.
    virtual int getArea() = 0;
    void setWidth(int w) {
      width = w;
    }

    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};

// Derived classes
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};

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

int main(void) {
  Rectangle Rect;
  Triangle Tri;

  Rect.setWidth(5);
  Rect.setHeight(7);

  // Print the area of the object.
```

```
   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;

   return 0;
}
```

Interface

C++ has no built-in concepts of interfaces. You can implement it using an abstract class which contains only pure virtual functions. Since it allows multiple inheritance, you can inherit this class to create another class which will then contain this interface (I mean, object interface :) in it.

Characteristics

1. An interface has no implementation.
2. An interface class contains only a virtual destructor and pure virtual functions.
3. An interface class is a class that specifies the polymorphic interface i.e. pure virtual function declarations into a base class. The programmer using a class hierarchy can then do so via a base class that communicates only the interface of classes in the hierarchy.

Implementation of Interface

```
#include<iostream>
Using namespace std;
class Base
{
Public:
   virtual void func1(int)=0;
   virtual void func1(int, int)=0;
};

class Derived : public Base
{
   public:
    virtual void func1(int someValue)
    {
        cout<<someValue;
     }

   private:
    virtual void func1(int someValue1, int someValue2)
   {
      cout<<0;
```

```
    }
};

class Derived2 : public Base
{
    public:
    virtual void func1(int someValue1, int someValue2)
    {
        cout<<someValue1+someValue2;
    }

    private:
     virtual void func1(int someValue)
     {

      }
 };

int main()
{
    Base *b;
    Derived d;

    b=&d;
    cout<<b->func(1,2); //this line would access the method even if it is private
return 1;
}
```


Interface vs. Abstract Class

An interface does not have implementation of any of its methods, it can be considered as a collection of method declarations. In C++, an interface can be simulated by making all methods as pure virtual. In Java, there is a separate keyword for interface.

Interfaces provide a convenient means of resolving the tension between what a class is and what it can do. Keeping interface and implementation separate in C++ programs keeps designs clean and fosters reuse.

Unit-IV

Polymorphism Introduction:

Polymorphism is another building block of object oriented programming. The philosophy that underlies is “one interface, multiple implementations.”
Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee. So same person posses have different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.
Polymorphism is derived from 2 greek words: **poly** and morphs. The word "poly" means many and **morphs** means forms. So polymorphism means many forms.
In programming languages, polymorphism means that some code or operations or objects behave differently in different contexts.

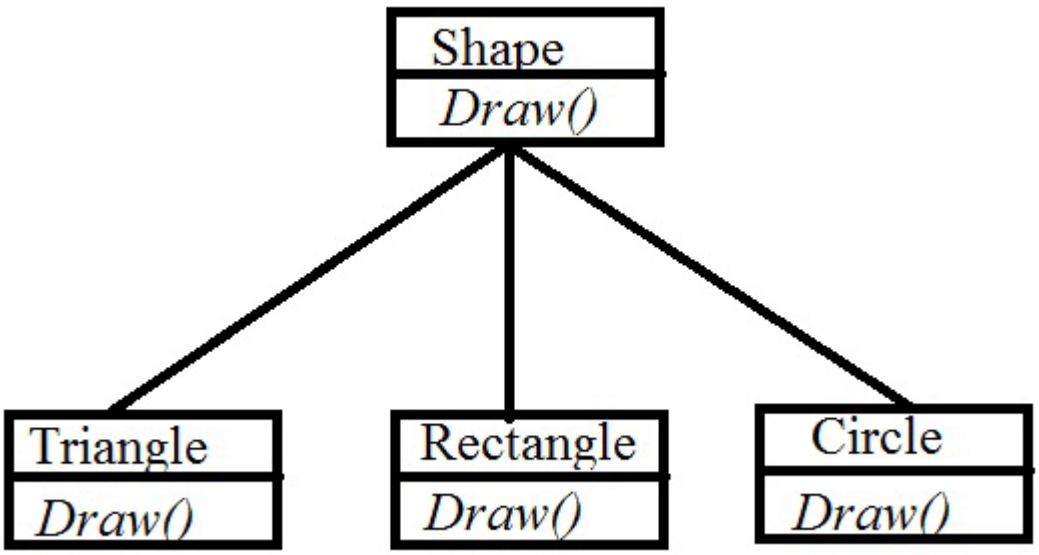


Fig 4.1 polymorphism

For example, the + (plus) operator in C++:

4 + 5 <-- integer addition
3.14 + 2.0 <-- floating point addition
s1 + "bar” <-- string concatenation!

Real life example of Polymorphism in C++

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person have different-different behaviors.

Usages and Advantages of Polymorphism

1. Method overloading allows methods that perform similar or closely related functions to be accessed through a common name. For example, a program performs operations on an array of

numbers which can be int, float, or double type. Method overloading allows you to define three methods with the same name and different types of parameters to handle the array of operations.

2. Method overloading can be implemented on constructors allowing different ways to initialize objects of a class. This enables you to define multiple constructors for handling different types of initializations.
3. Method overriding allows a sub class to use all the general definitions that a super class provides and add specialized definitions through overridden methods.
4. Method overriding works together with inheritance to enable code reuse of existing classes without the need for re-compilation.

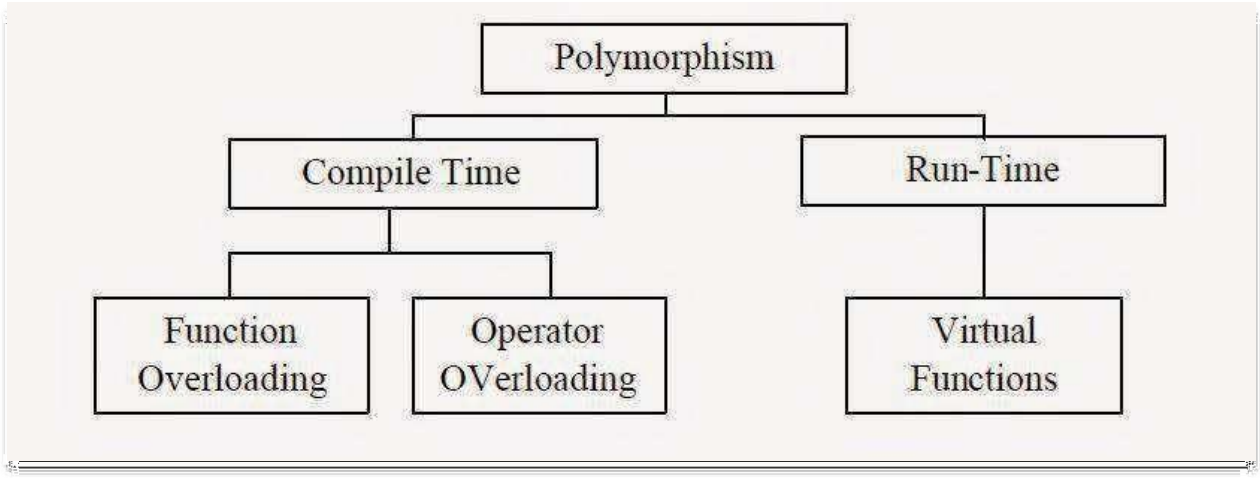


Fig 4.2 Polymorphism Type

Method Overloading and Overriding:

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**.

Why method Overloading

Suppose we have to perform addition of given number but there can be any number of arguments, if we write method such as a(int, int)for two arguments, b(int, int, int) for three arguments then it is very difficult for you and other programmer to understand purpose or behaviors of method they can not identify purpose of method. So we use method overloading to easily figure out the program. For example above two methods we can write sum(int, int) and sum(int, int, int) using method overloading concept.

Syntax

class class_Name

```
{
        Returntype method()
        {
        .....
        .....
        }
        Returntype  method(datatype1 variable1)
        {
        .....
        .....
        }
        Returntype  method(datatype1 variable1, datatype2 variable2)
        {
        .....
        .....
        }

};
```

Different ways to overload the method

- By changing number of arguments or parameters
- By changing the data type

By changing number of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.
Program Function Overloading in C++

```
#include<iostream.h>
#include<conio.h>

class Addition
{
public:
void sum(int a, int b)
{
        cout<<a+b;
}
void sum(int a, int b, int c)
{
        cout<<a+b+c;
}
};
void main()
{
      clrscr();
      Addition obj;
      obj.sum(10, 20);
```

```
        cout<<endl;
        obj.sum(10, 20, 30);
}
```

Output

30
60

By changing the data type

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two float arguments.

Method Overloading Program in C++

```
#include<iostream.h>
#include<conio.h>

class Addition
{
        public:
        void sum(int a, int b)
        {
        cout<<a+b;
        }
        void sum(float a, float b)
        {
                cout<<a+b+c;
        }
};
void main()
{
        clrscr();
        Addition obj;
        obj.sum(10, 20);
        cout<<endl;
        obj.sum(10, 20, 30);
}
```

Method Overriding

If derived class defines same method as defined in its base class, it is known as method overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the method which is already provided by its base class.

C++ Method Overriding Example

Let's see a simple example of method overriding in C++. In this example, we are overriding the eat() function.

```
#include  <iostream>
using namespace std;
class Animal
{
        public:
            void eat()
            {
                cout<<"Eating...";
            }
};
class Dog: public Animal
{
      public:
            void eat()
            {
                cout<<"Eating bread...";
            }
};
int main()
{

  Dog d = Dog();
  d.eat();
  return 0;
}
```

Output:

Eating bread...

Difference between Method Overloading and Overriding

Context	Method Overloading	Method Overriding
Definition	In Method Overloading, Methods of the same class shares the same name but each method must have different number of parameters or parameters having different types and order.	In Method Overriding, sub classes have the same method with same name and exactly the same number and type of parameters and same return type as a super class.
Meaning	Method Overloading means more than one method shares the same name in the class but having different signature.	Method Overriding means method of base class is re-defined in the derived class having same signature.
Behavior	Method Overloading is to "add" or "extend" more to method's behavior.	Method Overriding is to "Change" existing behavior of method.

Polymorphism	It is a compile time polymorphism.	It is a run time polymorphism.
Inheritance	It may or may not need inheritance in Method Overloading.	It always requires inheritance in Method Overriding.
Signature	In Method Overloading, methods must have different signature.	In Method Overriding, methods must have same signature.
Relationship of Methods	In Method Overloading, relationship is there between methods of same class.	In Method Overriding, relationship is there between methods of super class and sub class.
Criteria	In Method Overloading, methods have same name different signatures but in the same class.	In Method Overriding, methods have same name and same signature but in the different class.
No. of Classes	Method Overloading does not require more than one class for overloading.	Method Overriding requires at least two classes for overriding.

Static and Run Time Polymorphism

Static polymorphism: Static polymorphism refers to an entity existing in different physical forms simultaneously. Static polymorphism involves binding of functions based on the number, type, and sequence of arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to calls at compile time. This form of association is called early binding. The term early binding stems from the fact that when the program is executed, the calls are already bound to the appropriate functions.

The resolution of a function call is based on number, type, and sequence of arguments declared for each form of the function. Consider the following function declaration:

```
void add(int , int);
void add(float, float);
```

When the add() function is invoked, the parameters passed to it will determine which version of the function will be executed. This resolution is done at compile time.

Example of Static Polymorphism

1. Method Overloading
2. Operator Overloading

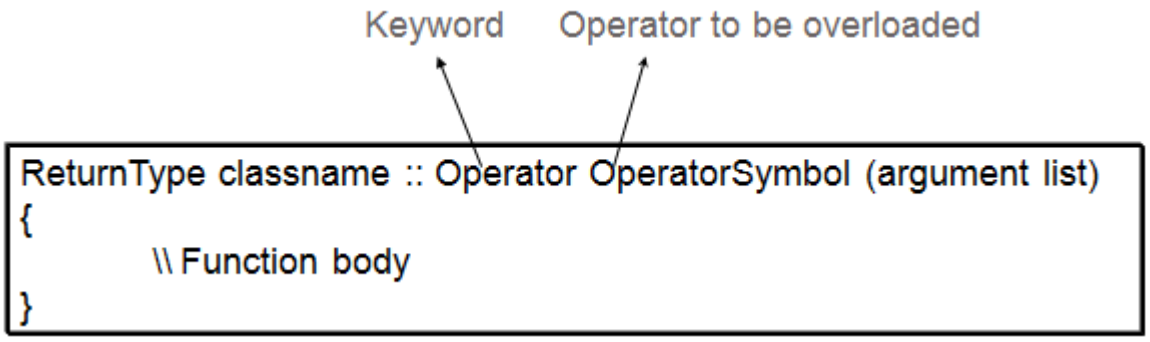
Operator Overloading

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String (concatenation) etc.

Operators that are not overloaded are follows

- scope operator - ::
- sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

Operator Overloading Syntax



Implementing Operator Overloading

Operator overloading can be done by implementing a function which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.

Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Restrictions on Operator Overloading

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added

There are two types of operator overloading in C++

- Binary Operator Overloading
- Unary Operator Overloading

Binary Operator Overloading Example

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

class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }
      void setLength( double len ) {
         length = len;
      }
      void setBreadth( double bre ) {
         breadth = bre;
      }
      void setHeight( double hei ) {
         height = hei;
      }

      // Overload + operator to add two Box objects.
      Box operator+(const Box& b) {
         Box box;
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
         return box;
      }

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

// Main function for the program
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);
```

```
   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

Run Time Polymorphism

Run Time Polymorphism: Dynamic polymorphism refers to an entity changing its form depending on the circumstances. A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the program is executed. The term late binding refers to the resolution of the functions at run-time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context.

Example of Run time Polymorphism

1. Method Overriding.
2. Virtual Functions.

Static Vs Dynamic Polymorphism

- Static polymorphism is also known as early binding and compile-time polymorphism.
- Dynamic polymorphism is also known as late binding and run-time polymorphism.
- Static polymorphism is considered more efficient and dynamic polymorphism more flexible.
- Statically bound methods are those methods that are bound to their calls at compile time. Dynamic function calls are bound to the functions during run-time. This involves the additional step of searching the functions during run-time. On the other hand, no run-time search is required for statically bound functions.
- As applications are becoming larger and more complicated, the need for flexibility is increasing rapidly. Most users have to periodically upgrade their software, and this could become a very tedious task if static polymorphism is applied. This is because any change in requirements requires

a major modification in the code. In the case of dynamic binding, the function calls are resolved at run-time, thereby giving the user the flexibility to alter the call without having to modify the code.

- To the programmer, efficiency and performance would probably be a primary concern, but to the user, flexibility or maintainability may be much more important. The decision is thus a trade-off between efficiency and flexibility

Virtual Function

A virtual function a member function which is declared within base class and is re-defined (Override) by derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Example of virtual function

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

    class BaseClass
    {

          public:
           virtual void Display()
           {
                cout<<"\n\tThis is Display() method of Base Class";
           }
           void Show()
           {
                cout<<"\n\tThis is Show() method of Base Class";
           }

    };
    class DerivedClass : public BaseClass
    {
          public:
           void Display()
           {
                cout<<"\n\tThis is Display() method of Derived Class";
           }
           void Show()
           {
                cout<<"\n\tThis is Show() method of Derived Class";
           }

    };
```

```
int main()
{

    DerivedClass D;
    BaseClass *B;          //Creating Base Class Pointer
    B = new BaseClass;
    B->Display();          //This will invoke Display() method of Base Class
    B->Show();             //This will invoke Show() method of Base Class
    B=&D;
    B->Display();          //This will invoke Display() method of Derived Class
                           //bcoz Display() method is virtual in Base Class

    B->Show();             //This will invoke Show() method of Base Class
                          //bcoz Show() method is not virtual in Base Class
   Return 0;
}
```

Output :

```
This is Display() method of Base Class
This is Show() method of Base Class
This is Display() method of Derived Class
This is Show() method of Base Class
```

Rules for Virtual Functions

1. They must be declared in public section of class.
2. Virtual functions cannot be static and also cannot be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
4. The prototype of virtual functions should be same in base as well as derived class.
5. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
6. A class may have virtual destructor but it cannot have a virtual constructor.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.
We can change the virtual function area() in the base class to the following −

```
class Shape
{
    protected:
        int width, height;

   public:
```

```
    Shape(int a = 0, int b = 0)
    {
       width = a;
       height = b;
    }

   // pure virtual function
    virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

Object Oriented Programming Systems
(CS-304)
Class Notes

Unit-V

String

String is a collection of characters. There are two types of strings commonly used in C++ programming language:

- Strings that are objects of string class (The Standard C++ Library string class)
- C-strings (C-style Strings)

C-strings

In C programming, the collection of characters is stored in the form of arrays, this is also supported in C++ programming. Hence it's called C-strings.

C-strings are arrays of type char terminated with null character, that is, \0 (ASCII value of null character is 0).

How to define a C-string?

```
char str[] = "C++";
```

In the above code, str is a string and it holds 4 characters.

Although, "C++" has 3 character, the null character \0 is added to the end of the string automatically.

Alternative ways of defining a string

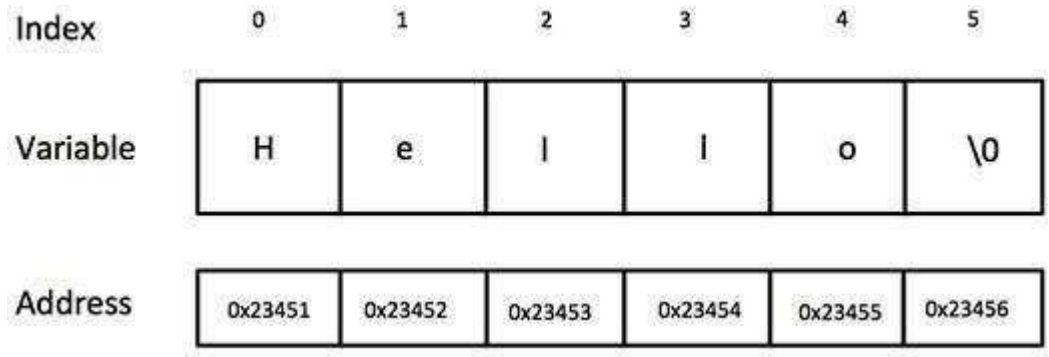
```
char str[4] = "C++";
char str[] = {'C','+','+','\0'};
char str[4] = {'C','+','+','\0'};
```


Like arrays, it is not necessary to use all the space allocated for the string. For example:

```
char str[100] = "C++";
```

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C/C++ −



Example 1: C++ String to read a word

C++ program to display a string entered by user.

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

```
int main()
{
   char str[100];

   cout << "Enter a string: ";
   cin >> str;
   cout << "You entered: " << str << endl;
   cout << "\nEnter another string: ";
   cin >> str;
   cout << "You entered: "<<str<<endl;
   return 0;
}
```

Output

```
Enter a string: C++
You entered: C++

Enter another string: Programming is fun.
You entered: Programming
```

Example 2: C++ String to read a line of text

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

int main()
{
   char str[100];
   cout << "Enter a string: ";
   cin.get(str, 100);

   cout << "You entered: " << str << endl;
   return 0;
}
```


Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

String Object

In C++, you can also create a string object for holding strings.
Unlike using char arrays, string objects has no fixed length, and can be extended as per your requirement.

Example 3: C++ string using string data type

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

int main()
```

```
{
   // Declaring a string object
   string str;
   cout << "Enter a string: ";
   getline(cin, str);

   cout << "You entered: " << str << endl;
   return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

Passing String to a Function

Strings are passed to a function in a similar way arrays are passed to a function.

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

void display(char s[]);

int main()
{
   char str[100];
   string str1;
   cout << "Enter a string: ";
   cin.get(str, 100);
   cout << "Enter another string: ";
   getline(cin, str1);
   display(str);
   display(str1);
   return 0;
}
void display(char s[])
{
   cout << "You entered char array: " << s <<;
}
void display(string s)
{
   cout << "You entered string: " << s << endl;
}
```


Output

```
Enter a string: Programming is fun.
Enter another string: Programming is fun.
You entered char array: Programming is fun.
```

You entered string: Programming is fun.

C++ supports a wide range of functions that manipulate null-terminated strings −

Sr.No	Function & Purpose
1	**strcpy(s1, s2);**Copies string s2 into string s1.
2	**strcat(s1, s2);**Concatenates string s2 onto the end of string s1.
3	**strlen(s1);**Returns the length of string s1.
4	**strcmp(s1, s2);**Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);**Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);**Returns a pointer to the first occurrence of string s2 in string s1.

```
#include <iostream>
#include <cstring>

using namespace std;

int main () {
   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int len ;
   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;
   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;
   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;

   return 0;
}
```


Output

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

The String Class in C++

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example −

```
#include <iostream>
```

```
#include <string>

using namespace std;

int main () {

   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int len ;

   // copy str1 into str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;

   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;

   // total length of str3 after concatenation
   len = str3.size();
   cout << "str3.size() :  " << len << endl;

   return 0;
}
```


When Should I Use std::string?

The advantages to using std::string:

- Ability to utilize SBRM design patterns
- The interfaces are much more intuitive to use, leading to less chances of messing up argument order
- Better searching, replacement, and string manipulation functions (c.f. the cstring library)
- The size/length functions are constant time (c.f. the linear time strlen function)
- Reduced boilerplate by abstracting memory management and buffer resizing
- Reduced risk of segmentation faults by utilizing iterators and the at() function
- Compatible with STL algorithms

Exception Handling

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Errors can be broadly categorized into two types. We will discuss them one by one.

1. Compile Time Errors
2. Run Time Errors

Compile Time Errors – Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import.

Run Time Errors - They are also known as exceptions. An exception caught during run time creates serious issues.

Why Exception Handling?

Following are main advantages of exception handling over traditional error handling.

1) Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

2) Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.
In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

3) Grouping of Error Types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Exception handling is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the system. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this we'll introduce exception handling techniques in our code.

In C++, Error handling is done by three keywords:-

- Try
- Catch
- Throw

Syntax

```
Try
{
//code
throw parameter;
}
catch(exceptionname ex)
{
//code to handle exception
}
```

Try

Try block is intended to throw exceptions, which is followed by catch blocks. Only one try block.

Catch

Catch block is intended to catch the error and handle the exception condition. We can have multiple catch blocks.

Throw

It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A throw expression accepts one parameter and that parameter is passed to handler.

Example of Exception
below program compiles successful but the program fails during run time.

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
        int a=10,b=0,c;
        c=a/b;
        return 0;
}
```


Implementation of try-catch, throw statement

Below program contains single catch statement to handle errors.

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
        int a=10,b=0,c;
        try //try block activates exception handling
        {
                if(b==0)
                throw "Division by zero not possible";//throws exception
                c=a/b;
}
catch(char* ex)//catches exception
{
        cout<<ex;
}
        getch();
        return 0;
```

```
}
```

Output

0

Example for multiple catch statement

Below program contains multiple catch statements to handle exception.

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
        int x[3]={-1,2,};
        for(int i=0;i<2;i++)
        {
                int ex=x[i];
                try {
                        if (ex > 0)
                                throw ex;
                        else
                                throw 'ex';
                        } catch (int ex) {
                                cout << " Integer exception;
                        }
                        catch (char ex)
                        {
                                cout << " Character exception" ;
                        }
                }
}
```

Output

Integer exception Character exception

Example for generalized catch statement

Below program contains generalized catch statement to catch uncaught errors. Catch(…) takes care of all type of exceptions.

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
        int x[3]={-1,2,};
        for(int i=0;i<2;i++)
        {
                int ex=x[i];
```

```
                try
                {
                if (ex > 0)
                        throw x;
                else
                        throw 'ex';
                }
                catch (int ex)
                {
                        cout << " Integer exception" ;
                }
                catch (char ex)
                {
                        cout << " Character exception" ;
                }catch (...)
                {
                        cout << "Special exception";
                }
        }
                return 0;
}
```

Output

Integer exception Special exception

Standard Exceptions in C++

There are standard exceptions in C++ under <exception> which we can use in our programs. They are arranged in a parent-child class hierarchy which is depicted below:

- **std::exception** - Parent class of all the standard C++ exceptions.
- **logic_error** - Exception happens in the internal logical of a program.
    - **domain_error** - Exception due to use of invalid domain.
    - **invalid argument** - Exception due to invalid argument.
    - **out_of_range** - Exception due to out of range i.e. size requirement exceeds allocation.
    - **length_error** - Exception due to length error.
- **runtime_error** - Exception happens during runtime.
    - **range_error** - Exception due to range errors in internal computations.
    - **overflow_error** - Exception due to arithmetic overflow errors.
    - **underflow_error** - Exception due to arithmetic underflow errors
- **bad_alloc** - Exception happens when memory allocation with new() fails.
- **bad_cast** - Exception happens when dynamic cast fails.
- **bad_exception** - Exception is specially designed to be listed in the dynamic-exception-specifier.
- **bad_typeid** - Exception thrown by typeid.

Define New Exceptions

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way –

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

struct MyException : public exception {
  const char * what () const throw () {
    return "C++ Exception";
  }
};

int main() {
  try {
    throw MyException();
  } catch(MyException& e) {
    std::cout << "MyException caught" << std::endl;
    std::cout << e.what() << std::endl;
  } catch(std::exception& e) {
    //Other errors
  }
}
```


This would produce the following result −

```
MyException caught
C++ Exception
```

Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

Multithreading in C++

Multithreading support was introduced in C+11. Prior to C++11, we had to use POSIX threads or p threads library in C. While this library did the job the lack of any standard language provided feature-set caused serious portability issues. C++ 11 did away with all that and gave us **std::thread**. The thread classes and related functions are defined in the **thread** header file.

std::thread is the thread class that represents a single thread in C++. To start a thread we simply need to create a new thread object and pass the executing code to be called (i.e, a callable object) into the constructor of the object. Once the object is created a new thread is launched which will execute the code specified in callable.

A callable can be either of the three

- A function pointer

- A function object
- A lambda expression

After defining callable, pass it to the constructor.

What is multi-threaded programming?

Single-threaded programs execute one line of code at a time, then move onto the next line in sequential order (except for branches, function calls etc.). This is generally the default behaviour when you write code. Multi-threaded programs are executing from two or more locations in your program at the same time (or at least, with the illusion of running at the same time). For example, suppose you want to perform a long file download from the internet, but don't want to keep the user waiting while this happens. Imagine how inconvenient it would be if we couldn't browse other web pages while waiting for files to download! So, we create a new thread (in the browser program for example) to do the download. In the meantime, the main original thread keeps processing mouse clicks and keyboard input so you can continue using the browser. When the file download completes, the main thread is signaled so it knows about it and can notify the user visually, and the thread performing the download closes down.

How does it work in practice?

Most programs start off running in a single thread and it is up to the developer to decide when to spin up (create) and tear down (destroy) other threads. The general idea is:

1. Application calls the system to request the creation of a new thread, along with the thread priority (how much processing time it is allowed to consume) and the starting point in the application that the thread should execute from (this is nearly always a function which you have defined in your application).
2. Main application thread and secondary thread (plus any other created threads) run concurrently
3. When the main thread's work is done, or if at any point it needs to wait for the result of a secondary thread, it waits for the relevant thread(s) to finish. This is (misleadingly) called a join operation.
4. Secondary threads finish their work and may optionally signal to the main thread that their work is done (either by setting a flag variable, or calling a function on the main thread)
5. Secondary threads close down
6. If the main thread was waiting on a join operation (see step 3), the termination of the secondary threads will cause the join to succeed and execution of the main thread will now resume

Threads

A **Thread** is a part of our software (Code Segment) that can run independently of the rest of the process. For example, one process with two threads will share the Code Segment, Data Segment and Heap. Much of the Process State can be shared as well; however, each thread will need some thread-specific state information held aside. In addition, each thread needs its own Stack space to keep track of the function and method calls.

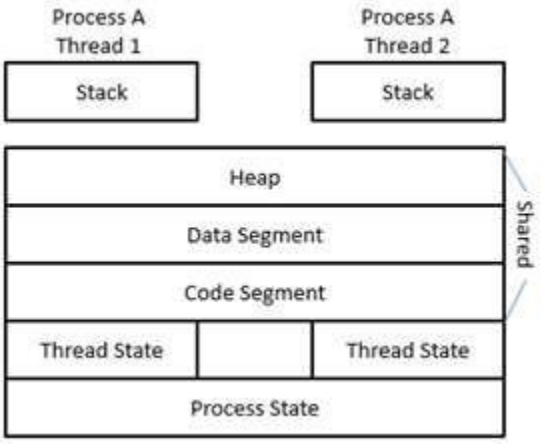


Fig. 5.1 Thread

Why create threads?

There are a few main reasons:

- You need to perform a task which takes a long time to complete but don't want to make the user wait for it to finish. This is called task parallelism and the threads which perform the long-running task are usually called worker threads. The purpose of creating the worker thread or threads is to ensure the application maintains responsiveness in the user interface, rather than to actually make the task run faster. Importantly, the task must be able to run largely independently of the main thread for this design pattern to be useful.
- You have a complex task which can gain a performance advantage by being split up into chunks. Here you create several worker threads, each dedicated to one piece of the task. When all the pieces have completed, the main thread aggregates the sub-results into a final result. This pattern is called the parallel aggregation pattern. See notes about multi-core processors below. For this to work, portions of the total result of the task or calculation must be able to be calculated independently – the second part of the result cannot rely on the first part etc.
- You want to serve possibly many users who need related but different tasks executing at the same time, but the occurrence time of these tasks is unpredictable. Classic examples are a web server serving many web pages to different users, or a database server servicing different queries. In this case, your application creates a set of threads when it starts and assigns tasks to them as and when they are needed. The reason for using a fixed set of threads is that creating threads is computationally expensive (time-consuming), so if you are serving many requests per second, it is better to avoid the overhead of creating threads every time a request comes in. This pattern is called *thread pooling* and the set of threads is called the thread pool.

This simple example code creates 5 threads with the pthread_create() routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit().

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;
```

```
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
   long tid;
   tid = (long)threadid;
   cout << "Hello World! Thread ID, " << tid << endl;
   pthread_exit(NULL);
}

int main () {
   pthread_t threads[NUM_THREADS];
   int rc;
   int i;

   for( i = 0; i < NUM_THREADS; i++ ) {
      cout << "main() : creating thread, " << i << endl;
      rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);

      if (rc) {
         cout << "Error:unable to create thread," << rc << endl;
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```


Data collection

A collection — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. They typically represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping from names to phone numbers).