Chapter 1

1 Object-Oriented Programming:

1.1 Structured Programming

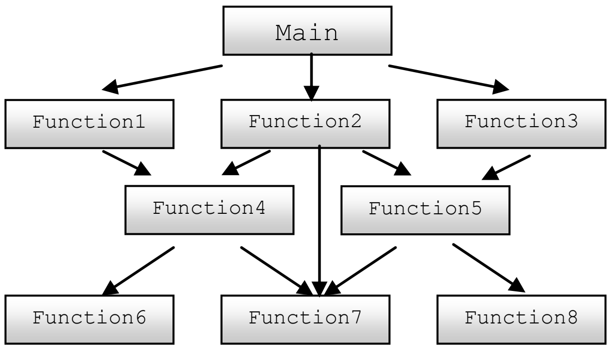
Given **problem is divided into sub problems** and each of these sub problems can be further divided and so on till each of these sub problems can be easily be coded

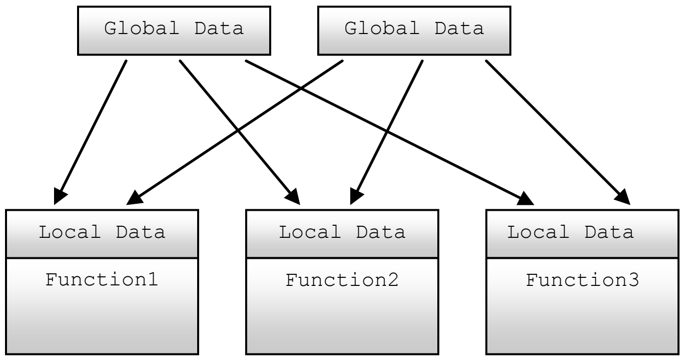
Each sub problem coded separately is called a **procedure** or **function**

Procedures and functions provide main importance in structured programming **but not data**

Many data items are placed as **global, accessible by all functions**

Global data are more vulnerable to an inadvertent change by a function





Emphasis is on doing things i.e. algorithms and not on data. Data is given second-class status while coding.

Any change in a data type being used needs to be made to all the functions using it . Time consuming.

In large programs it is very difficult to identify what data is used by which function.

Revising an external data structure means revising all functions that access the data .

Employs **Top-Down** approach i.e. first identify main function then other sub functions and then others.

It does not model the real world problems because functions are **action- oriented**.

1.2 Object Oriented Programming

Invented to remove **flaws encountered in procedural approach.**

Solves problem **in terms of objects used rather than procedures** or functions for solving it.

Gives **primary importance to data** being used instead of operations performed on the data.

The combination of **data** and its associated **functions** is known as an **object**.

Programs are divided into ***objects*** **not into functions**.

The close match between objects in the programming sense and objects in the real world helps us in thinking in terms of objects rather than functions that helps us to design programs easily.

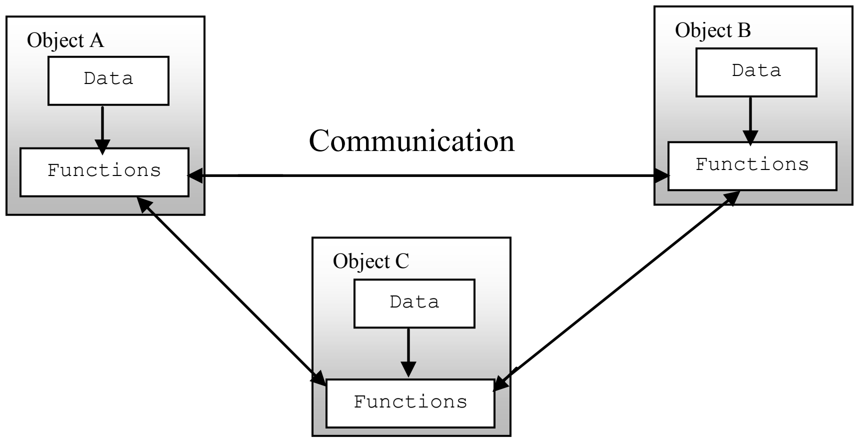
We can **access the data only thru the functions** associated with the object (protection).

New data and functions can be easily added whenever necessary.

Object is considered to be a partitioned **area of computer memory that stores data and set of operations** that can access that data.

Follows **Bottom-Up approach,** in program design**.**

"An approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand"
Identify objects in the system first and then identify their collaboration or interaction to meet the goal of the system
In an Object Oriented System, **the overall goal of the system is achieved thru the collaboration of objects**



1.3 Object-Oriented Programming as a New Paradigm

Paradigm: Any example, model, standard [Latin *paradigma*], overall strategy or approach to doing things

Programming Paradigms:

The **model of developing software** that describes the structure of computation

Imperative Programming: Pascal, C

Logic Programming: Prolog, Lisp

Functional Programming: FP, Haskell

"A programming paradigm is a **way of conceptualizing what it means to perform computation** and how tasks to be carried out on a computer should be structured and organized."

The style of problem solving embodied in *Object Oriented* technique is frequently the **method used to address problems in everyday life**

Computer novices are often able to grasp the basic ideas of OOP easily whereas computer literate people are often blocked by their own preconceptions.

Alan Kay (promoter of Object Oriented Paradigm, designer of Small Talk) found teaching Small Talk to children as being lot easier than to computer professionals

1.4 A Way of Viewing the World – Agents, Responsibility, Messages and Methods

Solving problem **mimicking the real world scenario**.

Real world analogy:

Sending flowers to a friend named Elise for her birthday

Mechanism used:

Find an appropriate ***agent*** (*namely Flo*)

Pass her a ***message*** containing the request (*kind and no. of flowers + address of friend*) for delivery of flowers

It is the ***responsibility*** of Flo now to satisfy the request

Flo uses some *method,* some algorithm or set of operations to do this

How the florist satisfies the request is not any of our concern.
Action is initiated in OOP by **transmission of a message to an agent (*an object*) responsible for action**

The **message** encodes the request for an action and is accompanied by any **additional information (*arguments*)** needed to carry out the request

The receiver is the agent to whom the message is sent

If the receiver accepts the message, it accepts the responsibility to carry out the indicated action

In response to a message the receiver will perform some method to satisfy the request.

e.g. while shopping you make a request to the shopkeeper, you don't show him how to get the things for you

```
    shopkeeper.giveMeJeans(color,length,waist,style);
      agent          message     arguments
```

Information Hiding

Client sending request need not know the actual means by which the request will be honored

Message Passing	Procedure Call
1. There is a **designated receiver** for that message	1. **No designated receiver**
2. ***Late binding*** between the message (function name) and the code fragment (method) used for response	2. ***Early binding*** (compile time) of name to code fragment
3. Interpretation of the message is dependent on the receiver and can vary with different receivers.	

1.4.1 Responsibility

Behavior is described in terms of ***responsibilities***, in OOP

For a particular request for action we can expect only the desired outcome i.e. a light bulb glows upon request (if switch is tuned on) but it doesn't make sound as the responsibility given to the light bulb is to glow (its behavior), it doesn't know how to make sound.

Responsibility permits greater independence between objects or agents (as each one can bear the responsibility without interference from others**)** e.g. a light bulb doesn't need others help from other objects to glow**,** it is functionally independent and capable**.**

Alan Kay's Description of Object Oriented Programming

i. Everything is an **object**

ii. Objects perform computation by making requests of each other thru the **passing of messages**

iii. Every object has it sown memory, which consists of other objects

iv. Every object is an **instance** of class. A **class** groups similar objects

v. The class is the repository of behavior associates with an object (**objects of a class perform same actions**)

vi. Classes are organized into singly rooted tree structure called an **inheritance hierarchy**

Basic concepts of OOP

i. Objects

Basic **runtime entities** that may represent a person, a place, a bank account etc

Programming problem is analyzed in terms of objects so they must be chosen such that they match closely with the real world objects

Objects **take up space in memory** and have an associated address like a structure in C

Objects **interact by sending messages** to one another during program execution
e.g. customer object requesting for the bank balance from account object, `account.getBalance(customerID);`

Each object has its name, data and code to manipulate the data (*functions*). In other words, each object is defined by its ***identity***, ***state*** and ***behavior***

Identity:

Each object can be identified by a **unique name**
What it is called. e.g. Bajaj fan

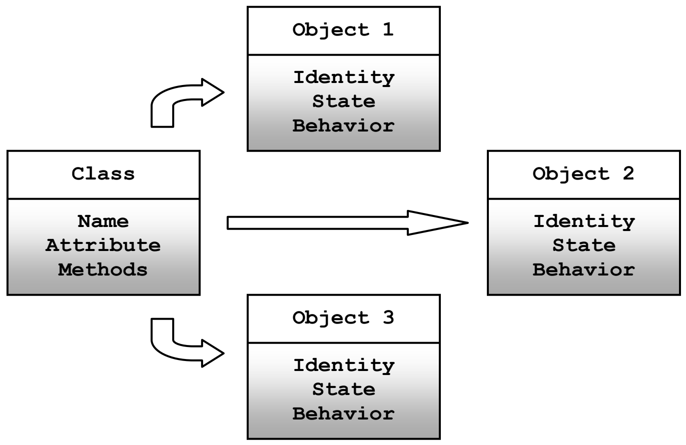
State:

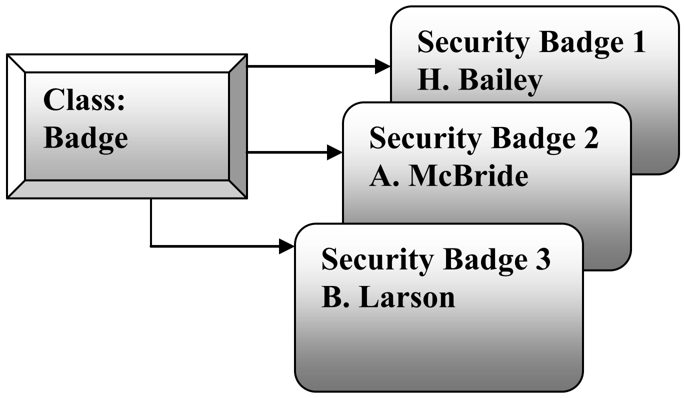
Each object maintains its state in terms of **variables**
What it is. e.g. Bajaj fan is in off state or on state

Behavior:

Each object implements its behavior with **methods (associated functions)**
What it does. e.g. Bajaj Fan can turn itself on or turn off upon request

Object	Identity	State	Behavior
Dog	Johnny	white, Doberman, hungry	sleeping, barking, eating
Pen	Parker	full, empty, nib out, nib in	write, refill, leak down
Cup	Coffee Cup	white , hot, full	can be filled, drunk from, carried
Fan	Khetan	white, big, off, on	turn on , turn off





In OOP objects are identified as

Tangible Things : physical objects e.g. car, bicycle, house, fan

Roles: of people or organization e.g. employee, account holder, employer

Incidents: Something happening at a particular point in time flight, transactions (deposit/withdrawal)

Interaction: a link between objects e.g. electrical connection

Specification: a definition of a set of objects e.g. Description of a particular type of stock item

ii. Class

A **collection of objects** that share common properties and relationships

An OO concept that **encapsulate the data and procedural abstractions** (*methods*) required to describe the contents (*state or attributes*) and behavior of some real world entity

A class is an abstract data-type, which acts as a ***blue print*** or a ***prototype*** that defines the variables and methods common to all objects of a certain kind.

Objects are variables of the type *class*

A class is comprised of a set of attributes and behavior shared by similar type of objects
e.g. mango, orange apple are members of the class fruit

Attributes

Elements that **make up the state** of that class objects

In programming terms they are known as *variables, data items*

Methods

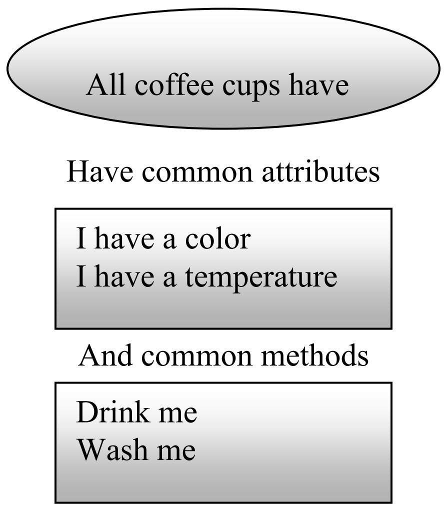
Methods act as the only path between the user and the state data

Class **data can only be accessed via these methods**
(*Data or Information Hiding*)

Types of methods

Selector Method
Read/ Get Method
Allows access to an attribute but does not allow that attribute to be changed
Analogous to functions (*data is unaffected by executing a function*)

Modifier Method
Write/Set Method
Allows attribute to be changed
Analogous to procedures (*data is affected by executing a function*)

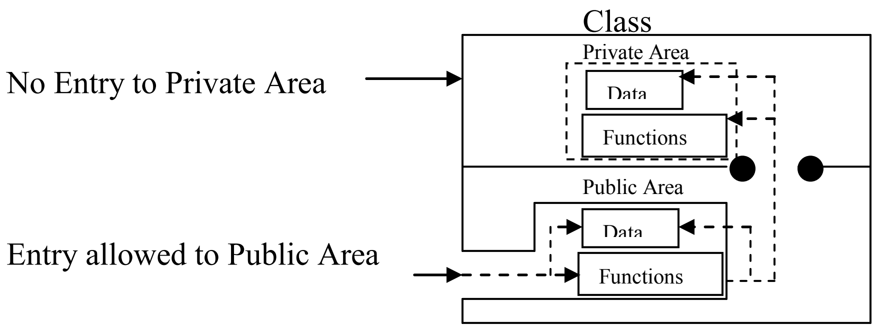


Data Abstraction and Encapsulation

The **wrapping up of data and function** into a single unit (*class*) is known as ***encapsulation.***

The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.

This insulation of the data from direct access by the program is called **data hiding or information hiding**



The act of **representing essential features with out including the back ground details** is known as ***abstraction***.

e.g. we use a switch board for switching on a fan or bulb, we need not know about the internal fitting or back ground details

Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight , cost and functions to operate on these attributes

Encapsulation is a way to implement ***data abstraction***

The ***attributes*** are called ***data members*** as they hold information

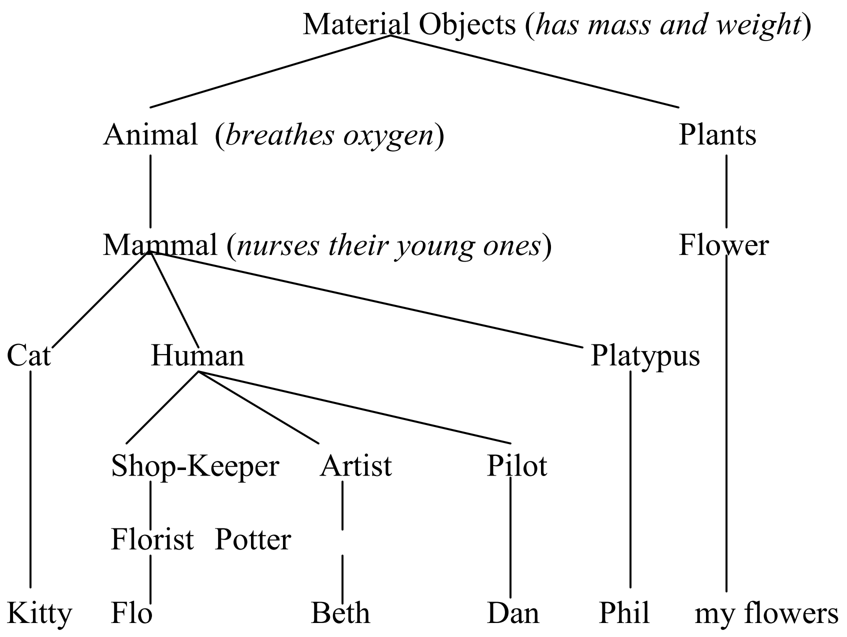
The ***functions*** that operate on these data are called ***member function*** or ***methods***

Since classes use the concept of data abstraction , they are known as *abstract data types* (*ADT*)

iii. Inheritance:

Inheritance is the process by which objects of one class **acquire the properties of objects of another class**

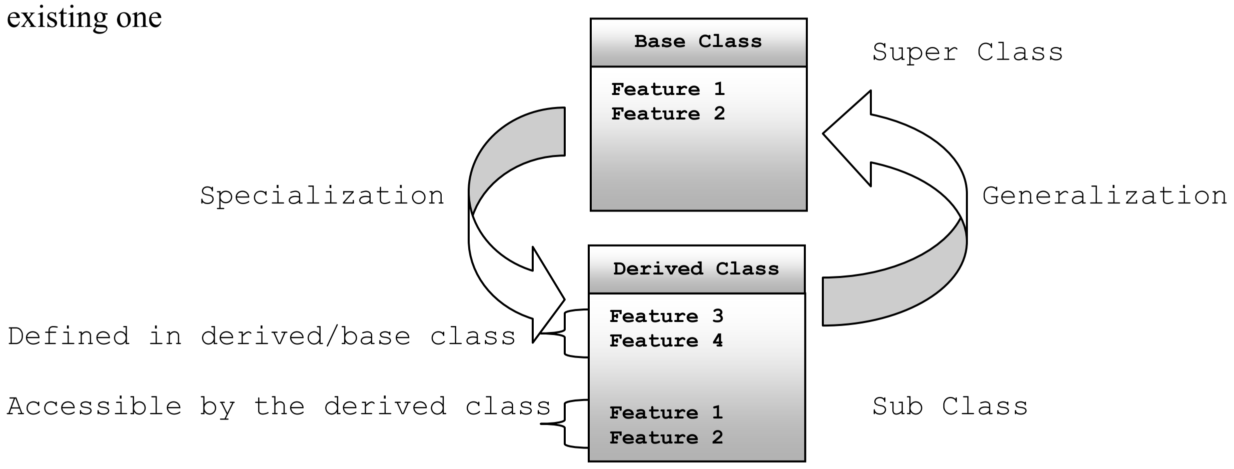
It supports the concept of ***hierarchical classification***



Material Objects (*has mass and weight*)
Animal (*breathes oxygen*)
Plants
Mammal (*nurses their young ones*)
Flower
Cat
Human
Platypus
Shop-Keeper
Artist
Pilot
Florist
Potter
Kitty
Flo
Beth
Dan
Phil
my flowers

It provides the idea of ***reusability***

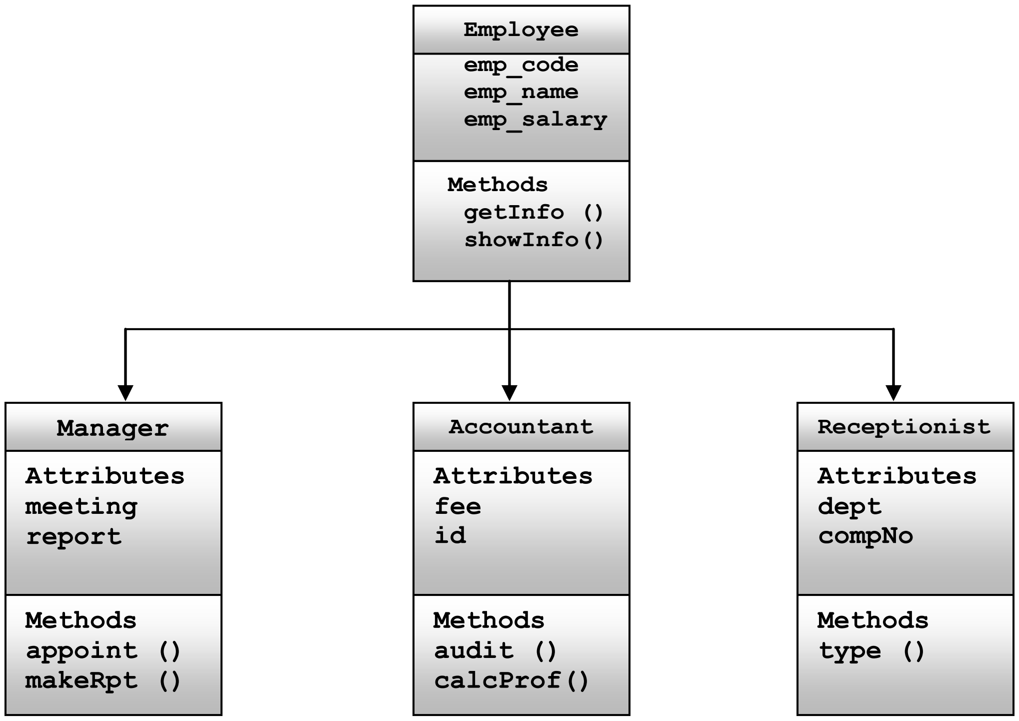
We can add additional features to an existing class without modifying it by deriving a new class from the existing one



The existing class is called the ***base class*** (*generalization of the newly derived class*) and the new class is called the ***derived class*** (*specialization of the existing base class*)

This new class will have the combined features of both the classes

In case of the employees of an organization, all employees share the common attributes *emp_salary, emp_code, emp_name*



iv. Polymorphism

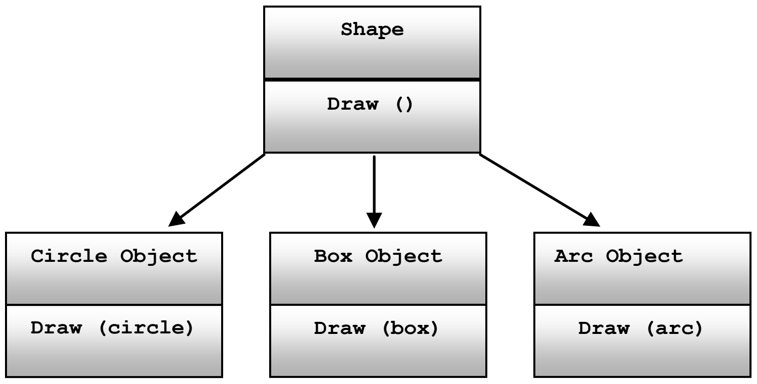
Poly means **many** and ***Morphos*** means **forms** , together it means the **ability to take more than one form**

Polymorphism means that it is possible to ***overload*** (*use to mean more than one thing*) the *symbols* (*operators and function names*) that we use in a program so that the **same symbol** can have different meanings in **different context**.

An operation may exhibit different behavior in different instances depending upon the types of data used in the operation

e.g. for the same request the Flo might respond in a different way (*sends flowers in a car*) than one of my friends (*he might use a motorcycle to drop those flowers*)

```
#include <iostream.h>
void display(char a[], char b[]){
  cout << strcat(a,b) ;
}
void display(int a, int b){
  cout << a+b ;
}
void main(){
 char a[]="Object", b[]="Oriented";
 int  c=99,d=100;
  display(c,d); display(a,b);
}
```



Similarly, operators +, -, = can also be used to mean more than one thing

The process of making an operator or function to exhibit different behavior in different instances is known as ***operator or function overloading*** respectively.

v. Dynamic Binding

Binding means **linking a procedure call to the code to be executed** in response to the call

Static Binding (Early Binding): If the **binding** of a function call to the function code **is done during compile time** it is known as ***static binding***

Dynamic Binding(Late Binding): Means that the **code associated with a given procedure call is not known until the time of the call at *run-time***. It is associated with *polymorphism* and *inheritance*.

In the above example, every object will inherit the *draw()* method from their parent class Shape

However, its algorithm is unique to each other as the draw procedure will be redefined in each class that defined that object

At run time the code matching the object under current reference will be called.

vi. Message Passing

Objects **communicate** with each other thru sending and receiving messages much the same way as people pass messages to one another

The difference is in case of objects the message needs to be more precise/specific

In case of people, we ask a driver to drive faster

In case of objects the object making the above request also needs to **specify the amount**

In case of a message passing, there is a desired **recipient** of the message.

1.5 Computation as Simulation

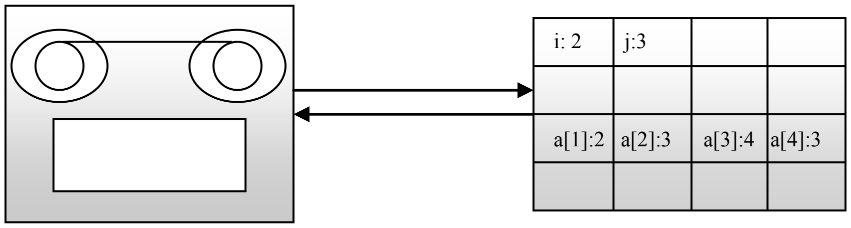
Process State/Pigeon Hole Model:

A traditional model describing the behavior of a computer executing a program

The computer is a **data manager** following some **pattern of instructions**, wandering thru **memory**, pulling **values** of various slots (memory address), transforming them in some manner and pushing the results back into other slots

Values in the slots describes the state of the machine or results of computation

But, this model doesn't **help us in understanding how to solve problems** using computer



Object Oriented Model:

No mentioning of memory addresses, variables, assignments at all.

Instead we **speak of objects, messages, responsibility** for some action

Animistic: Process of **creating a host of helpers** that form a community and assist the programmer in the solution of a problem

This view of programming as creating a universe is similar to style of computer simulation called ***Discrete Event Driven Simulation***, where the user **creates computer models of the various elements of simulation, describes how they will interact with one another and sets them moving**

The same approach is used in Object Oriented Program in which the user **describes what the various entities in the universe for the program are and how they will interact with each other and finally sets them in motion**.

In Object Oriented Programming, we have the view that **computation is simulation**

1.5.1 Power of Metaphor

Metaphor: Figure of speech, symbol

An *overlooked* benefit

When programmers think about problems in terms of **behaviors and responsibilities of objects they bring with them a wealth of intuition, ideas and understanding from their everyday experience**

When envisioned as pigeon holes, slots containing values there's little in the programmers background to provide insight into how problems should be structured

Easier to teach OOP concepts to computer novices as they quickly adapt the metaphors with which they are comfortable from their everyday life

Whereas, computer professionals are blinded by an adherence to the traditional ways of viewing computation

1.6 Complexity:

In the beginning, programs were written in assembly language by a single individual

As programs became complex it became hard to develop and debug their software

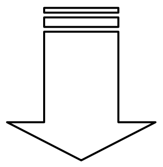
- Remembering which values stored in what registers
- Variable name conflicts
- Variable initialization decisions

Introduction of higher level languages like FORTRAN, Cobol solved some difficulties (*e.g. automatic management of local variables*)

But **it also rose people's expectations** of what a computer could do so programs started getting longer and more complex e.g.

Rise in Expectation **Rise in Complexity level**

A banking system that could automate a particular branch's tasks
A banking system that could perform tasks between different branches
of the same bank (intra banking)
A banking system that could perform tasks from any where (intranet based)
A banking system that could perform tasks using any devices or gadgets (mobile based)



Even the best programmers lost the track in an effort to solve complex problems using computers

1.6.1 The Nonlinear Behavior of Complexity:

A task that would take one programmer two months to perform could not be accomplished by two programmers working for one month (refer to Freed Brook's book Mythical Man-Month)

"The bearing of a child takes nine months, no matter how many women are assigned to the task"

Reason for *non linear behavior*:

- **Complexity**
- **Interconnections** between software components were complicated
- Large amount of **information had to be communicated among members** of the programming team

Conventional techniques brings about high degree of interconnectedness

This means the dependence of one portion of code on another portion (coupling)

So, a complete understanding of what is going on requires a knowledge of both portion of code we are considering and the code that uses it e.g. if you are considering a calendar class coded in C++, you also need to know about the rest of the code that uses that calendar class in that application.

An individual section of code cannot be understood in isolation

1.6.2 Abstraction Mechanisms

A mechanism used by programmers to control complexity

The ability to encapsulate and isolate **design** and **execution information**

i. Procedures

Procedures and *Functions* were two of the first abstraction mechanisms to be used in programming

Procedures and functions facilitate ***code reuse***

In addition, they gave the first possibility for ***information hiding***

Procedure or functions written by one programmer could be used by many other programmers

No need to know the exact *implementation details* they only needed the ***necessary interface***

```
 interface     ──────────►    float calculateEmpSalary(char * impName, float basicSalary, float bonus){
                                      --------          --------
implementation──────────────────►     --------          --------
                                      return(totalSalary);
                              }
```

However, procedures were **not an effective mechanism for information hiding**, they only partially solved the problem of multiple programmers using the same name

ii. Modules

An improved technique for creating and managing collection of names and their associated values

A module provides the ability to divide a name space into two parts.

The **public part** that is accessible outside the module and the **private** that is accessible only with the module

Follows the military doctrine philosophy, "*If you do not need to know some information, you should not have access to it*"

iii. Abstract Data types

The programmer defined data types that can be manipulated like the system defined data types

It corresponds to a set of legal data values and a number of primitive operations that can be performed on those values

Users can create variables of these data types and perform the permitted operations on them

1.7 Reusable Software

Viewing construction of software as being similar to construction of material objects e.g. car, building

Putting of pieces of off the shelf components rather than fabricating each new element from scratch

Code once, reuse many times e.g. a calendar module coded once can be used in many web sites many times

Chapter 2

2 Object-Oriented Design:

People often tend to compare the **syntactic features** of *C++ , Java* with non object oriented versions *C ,Pascal* in response to questions regarding Object Oriented Programming

Then about *classes, inheritance , message passing* etc

They are overlooking the crucial point of *Object Oriented Programming*

Object Oriented Programming has **nothing to do with *syntax***

The most important aspect of OOP is a **design technique driven by the determination and delegation of responsibilities.**

This technique is called ***Responsibility Driven Design (RDD).***

2.1 Responsibility Implies Noninterference:

When we make an object (*be it a child or software*) responsible for **specific actions** we expect a **certain behavior** e.g. a light bulb glowing (behavior) when it is switched on (action)

Responsibility implies *a degree of independence or non-interference,* i.e. an object is fully responsible for doing certain thing that it has taken as its responsibility. e.g. a light bulb object has the responsibility of glowing and it can do it on its own

When we assign a child the responsibility of cleaning her room, we expect that the desired outcome will be produced (*a clean room*)

Normally, we do not interfere nor do we stand over her and watch while that task is being performed

We leave it up to her to apply the method she desires to clean her room (use a vacuum cleaner, broom)

Conventional programming proceeds largely by doing something to something else e.g. modifying a record or updating an array

Here one portion of code in a software system is often intimately tied by control and data connections to many other sections of the system due to the use of global variables , use of pointers etc

A responsibility driven design attempts to cut these links

In case of, conventional programming we tend to actively supervise the child while she's performing a task

In case of OOP we tend to handover to the child responsibility for that performance

2.2 Programming in the Small and Programming in the Large:

The difference between the development of individual projects and of more sizeable software systems is described as *programming in the small versus programming in large*

Programming in small characterizes projects with the following attributes:

Code is developed by a **single programmer** or by a very small collection of programmers

A single individual can **understand all aspects of a project** form top to bottom, beginning to end

Major problem in the software development process is the design and development of algorithms for dealing with the problem at hand

Programming in large characterizes projects with the following attributes:

Software system is developed by a **large team of programmers**

Different team members involved in

- Specification or **design** of a system

- **Coding** of individual components
- **Integration** of various components in the final product

No single individual can be considered responsible for the entire project

No single individual understands all aspects of the project

Major problem in the software development process is the **management of detail** and the c**ommunication of information** between diverse portions of the project

2.3 Role of Behavior in OOP:

It is better to start the design process with an **analysis of behavior** as the behavior of a system is usually understood long before any other aspect

Earlier software development techniques concentrated on ideas such as characterizing the basic data structures or the overall structure of function calls of the desired application

- But structural elements of the application can be identified only after a considerable amount of problem analysis
- Similarly a formal specification often ended up as a document understood by neither programmer nor client

But behavior can be described almost from the moment an idea is conceived to both the programmers and the client e.g. player throws dice, **dice rolls** and returns a value and **result** is **declared** based on the comparison of the face value of the dice

In case of OOP, the design is based on responsibility (*Responsibility Driven Design*)

2.4 Case Study: *Responsibility-Driven Design*

2.4.1 Objective: Develop an Interactive Intelligent Kitchen Helper (*IIKH*)

2.4.2 Brief Description:

A PC based application to be designed to replace the index card system of recipes in the kitchen

Maintains a database of recipes and assists in the planning of meals for an extended period

User sits down at a terminal browses the database of recipes and interactively create a series of menus

The IIKH should automatically scale the recipes to any number of servings and print out menus for the entire week, for a particular day, or for a particular meal

Print out an integrated grocery list of all items needed for the recipes for the entire period

As with most software systems, initial descriptions of IIKH are highly ambiguous

The project will require the efforts of several programmers working together

The initial goal of the team must be to clarify the ambiguities in the description and to outline how the project can be divided into components to be assigned for development to individual team members

The fundamental cornerstone of Object-Oriented Programming is to characterize software in terms of behavior

Initially, the team will try to characterize the behavior of the entire application at a very high level of abstraction

This then leads to a description of the behavior of various software subsystems

The software design team proceeds to the coding step only when all behavior has been identified and described

2.4.3 Identification of Components:

Just as in case of generic engineering where simplification is achieved by dividing the design into smaller units

Similarly the engineering of software is simplified by the identification and development of software components

A component is simply **an abstract entity that can perform tasks** or fulfill responsibilities

A component may ultimately be turned into a function, a structure or class

Characteristics of components

- A component must have a small well defined **set of responsibilities**
- A component should **interact with other components to the minimal extent possible**

2.5 CRC Cards:

A CRC card is an **object oriented design method** that uses ordinary 3x5 index cards. It was developed by Ward Cunningham at Textronix, a card is made for each class containing responsibilities (knowledge and services) and collaborators (interaction with other objects)

The first step of system development is to **gather user requirements.**

We can't build a system if we don't know what it should do.

A CRC modeling provides a simple and effective technique for working with the users to determine their needs.

The team **discovers the components** and **their responsibilities by walking thru the scenario**

Every activity that must take place is identified and assigned to some component as a responsibility

Components are represented using small index cards with the name of the software component, its responsibilities and the names of other components it interacts with, written on the face.

These cards are known as CRC Cards (*Component, Responsibility, Collaborator*) and associated with each software component

Component Name	
Description of responsibilities assigned to this component.	**Collaborators** **List of other Components**

Collaborators:

Sometimes classes do not have enough information to fulfill their responsibilities, so they need to **work (*collaborate*) with other classes** to get the job done

A collaboration will be in either of the two forms:

- A request for information
- A request to perform a task

Class B is listed as a collaborator of Class A if and only if B does something to A

During scenario simulation, CRC cards can be used by assigning them to the members of the design team to give physical representation of the components.

Physical separation of the cards encourages an intuitive understanding of the importance of the logical separation of the various components helping to emphasize the cohesion and coupling

A component might be assigned more tasks than it can handle, in such case some responsibilities can be shared with other new components

2.5.1 The What/Who Cycle:

The identification of components takes place during the process of imagining the execution of a working system

This initiates the cycle of *what/who* questions

The programming team identifies **what activity** needs to be performed next

Then identifies **who** performs the action, **BUT NOT HOW, YET!!**

As in real world scenario, some component must be assigned a responsibility to perform an activity

The secret to good object oriented design is to first establish an agent for each action

2.5.2 Documentation:

At this point the development of documentation should begin.

Two types of documents as an essential part of any software system

i. **The User Manual**

Describes he interaction with the system from the user's point of view

It should be developed before any actual code has been written, the mindset of the team is similar to the eventual users at this point in time

ii. **The System Design Documentation**

Records the major decisions made during software design and should be produced when these decisions are fresh in the minds of the creators

Gives the initial picture of the larger structure hence should be carried out early in the development cycle

CRC cards are one aspect of the design documentation but not many other important decisions are not reflected in them

2.6 Components and Behavior:

2.6.1 Characteristics, behavior and responsibilities of Components identified for the IIKH System:

a. Greeter

The IIKH team decides that **system greets the user with an attractive informative window**.
The **responsibility for displaying this window** is assigned to a component called the *Greeter*

The user can select one of the five actions as designed and identified by the team:

i. Casually browse the database of existing recipes without reference to any particular meal plan

ii. Add a new recipe to the data base

iii. Edit or annotate an existing recipe

iv. Review an existing plan for several means

v. Create a new plan of meals

Greeter	
Display Informative Initial Message **Offer User Choice of Options** **Pass Control to either:** **Recipe Database Manager** **Plan Manager** **for processing**	**Collaborators** **Database Manager** **Plan Manager**

The whole activities can be divided into two groups.

i. First three as being associated with the recipe database

ii. The latter two are associated with the menu plans

The team decides to create components corresponding to these two responsibilities

b. **Recipe Component**

- Each recipe will be identified with a specific recipe component.
- Once a recipe is selected control is passed to the associated recipe object
- It consists of a list of ingredients and steps needed to transform the ingredients into the final product
- It should display the recipe interactively on the screen
- It should enable the user to modify the instructions or list of ingredients
- It gives printed copy of the recipe

c. **Recipe Database Manager**

In response to the user's desire to add a new recipe the database manager has to decide in which category to place the new recipe

It is done by requesting the name of the new recipe and then creating a new recipe component permitting the user to edit the new blank entry, a subset of tasks already identified for allowing the user to exit existing recipes

These task handle the browsing and creation of new recipes

Now we investigate the development of daily menu plans which is the plan manager's task

d. **Plan Manager**

Can either be started by retrieving an already developed plan or by creating a new plan (use can specify the date)

Each date is associated with a date component

Print out the recipe for the planning period, produce grocery list for the period

e. **Date Component**

Maintain a collection of meals as well as any other remarks provided by the user (anniversary, birthday)

Out puts information concerning the specific date

Allows the user to print all the information concerning a specific date find out more about a specific meal (control is passed to the meal component in this case)

f. **Meal Component**

Displays information about the meal

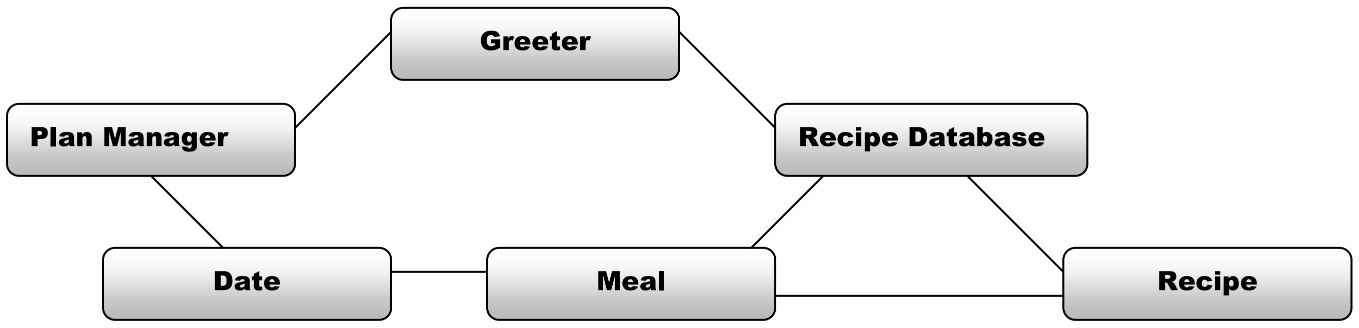
Maintains a collection of augmented recipes (user's desire to double , triple or increase a recipe)

The user can add or remove recipes from the meal and print meal information

It must interact with database component to allow user to discover new recipes by browsing the recipe database

In this manner the design team investigate every possible scenario

Exceptional cases like no matching recipe found for the user input, user wants to cancel an activity etc must be handled by assigning the task to some component.



Communication Between the Six Components in the IIKH

The team analyzed that all activities can be adequately handled by six components

- The *Greeter* needs to communicate only with the *Plan Manager* and the *Recipe Database* component
- The *Plan Manager* needs to communicate only with the *Date* Component
- The *Date Agent* needs to communicate only with the *Meal* Component
- The *Meal* component communicates with individual recipes thru the *Recipe* Manager

2.6.2 Preparing for a Change:

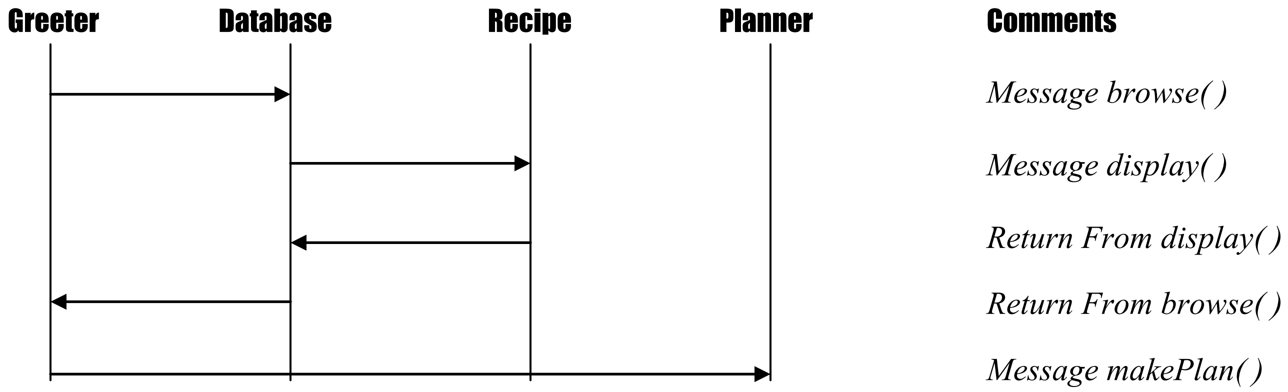
Changes in user's needs or requirements will force changes to be made in the software

Programmers need to anticipate this and plan accordingly

- Changes should affect as few components as possible
- Predict most likely sources of change and isolate the effects of such changes to as few components as possible
- Reduce coupling between software components (*will reduce the dependency of one upon another*)
- Isolate and reduce the dependency of software on hardware
- In design documentation, maintain careful records of the design process and discussions on decisions

2.6.3 Interaction Diagram

Used for describing the **dynamic interaction between components** during the execution of a scenario



Here, time moves forward from the top to bottom.

Each component is represented by a labeled vertical line.

A component sending a message to another is represented by a horizontal arrow from one line to another

Similarly a component returning control or a result value back to the called is represented by an arrow

Commentary on the right side of the figure fully explains the interaction taking place

It describes the sequencing of events during a scenario with a time axis, so can be used as a documentation tool for complex software systems

2.7 Software Components:

Each component is characterized by:

i. **Behavior**

- Set of actions it can perform (*what they can do*)
- The complete description of all the behavior for a component is called a *protocol*

- For Recipe component, this includes activities like editing the preparation instructions, displaying the recipe on the screen, printing the recipe

ii. State

- Each component holds certain information
- For Recipe component, the state includes the ingredients and preparation instructions
- State is not static as it can change over time
- The user can make changes to the preparation instructions (*state*) by editing a recipe (*behavior*)

2.7.1 Instances and Classes:

In case of a real application, there will be many different recipes.

However, all of these recipes will perform in the same manner i.e. behavior of each recipe is same

But the state (*e.g. individual list of ingredients, instructions etc*) differs between individual recipes

Class refers to a set of objects with similar behavior

An individual representation of class is known as an *instance*

All instances of a class respond to the same instructions and perform in a similar manner

But these instances all may be in different state as state is the property of an individual

2.7.2 Coupling and Cohesion:

Cohesion is a qualitative indication of the degree to which a module focuses on just one thing

A *cohesive module* should do just one thing

High cohesion is achieved by associating in a single component tasks that are related in some manner

Coupling is a qualitative indication of the degree to which a module is connected to other modules an to the outside world

The amount of coupling should be reduced as connections between software components decrease ease of development, modification and reuse (*ripple effect*)

Coupling is increased when one software component must access data values (the state) held by another component

This situation should be avoided as far as possible

For example, in case Recipe Database in IIKH, while performing tasks associated with this component the need to edit a recipe first occurs

So it would be wise to assign responsibility for editing a recipe first

2.7.3 Interface and Implementation:

The deliberate omission of implementation details behind a simple interface is known as information hiding

It is possible for one programmer to know how to use a component developed by another programmer with out needing to know how the component is implemented

The component encapsulates the behavior showing only how the component can be used and not the detailed actions it performs

Two views of a Software System:

i. Interface View

- The face seen by other programmers
- Describes what a software component can perform

ii. **Implementation View**

- The face seem by the programmer working on a particular component
- Describes how a component goes about completing a task

The separation of interface and implementation is the most important concept in software engineering

Information hiding is largely meaningful only in context of multi person programming projects

Here, the limiting factor is not the amount of coding involved but the amount of communication required between programmers and their software systems

For the reuse of general purpose software components in multiple projects there must be minimal and well understood interconnections between the various portion so the system

Parnas's Principles (**David Parnas**):

- The developer of a software component **must provide the intended user with all information needed** to make effective use of the services provided by the component and should provide no other information
- The developer of a software component **must be provided with all the information necessary to carry out the given responsibilities** assigned to the component and should be provided with no other information

2.8 Formalize the Interface:

This step includes making decision regarding the general structure that will be used to implement each component:

i. A component with only **one behavior and no internal state** may be made into a **function** e.g. a component that takes a string of text and translates all capital letter to lowercase

ii. Components with **many tasks responsibilities** are more easily implemented as **classes**, use a CRC card to record all responsibilities and convert them into procedures or functions

iii. Along with the **names**, the **types of arguments to be passed** to the function are identified

iv. Next, the information maintained with in the component itself should be described in detail
e.g. source of data must be clearly identified i.e. either local , global or passed thru an argument

2.8.1 Coming up with names

- Names associated with various activities should be carefully chosen
- Names create the vocabulary with which the eventual design will be formulated
- Names should be meaningful, short and suggestive in the context of the problem

General guidelines for choosing names:

- Use pronounceable names (*one that you can read out loud*)
- Use capitalization (*or underscores*) to the beginning of a new word with in a name "*CardReader*" or "*Card_Reader*"
- Abbreviations should not be confusing
  TermProcess: a terminal process or something that terminates a process?
- Avoid names with many interpretations. e.g. empty(), full()
- Avoid digits within a name , 1 and l, 2 and Z , 0 and o
- Name functions and variables that yield *Boolean values* so that they describe clearly the interpretation of a true or false value. e.g. *PrinterIsReady* --> a true value means the printer is working where as *PrinterStatus* is less precise

- Names for operations that are costly and infrequently used should be carefully chosen as this can avoid errors caused by using the wrong function

The CRC cards for each component, along with the name and formal arguments of the function depicting its behavior, are redrawn after the names have been developed for each activity

Date	
Maintain information about specific date *Date(year, month, day)* **–create new date** *DisplayAndEdit( )*- **display date information in window allowing user to edit entries** *BuildGroceryList(List &)*- **add items from all means to grocery list**	**Collaborators** **Meal Manager** **Meal**

Revised CRC card for the Date Component

2.9 Design the Representation for Components:

The design team is divided into groups each responsible for one or more software component

Transformation of the description of a component into a software system implementation is done now

Designing of the data structures that will be used by each subsystem to maintain the state information required to fulfill the assigned responsibilities is done

Once the data structure is chosen (an important task) the code used by the component in the fulfillment of a responsibility is often self evident

A wrong choice can lead to complex and inefficient program

Now the description of behavior must be transformed into algorithms

These descriptions should fulfill the expectations of each collaborator component

2.10 Implementing Components:

Each component's desired behavior is implemented in a computer language after the design phase is over

Proper design yields a short description of each responsibility or behavior

As one programmer will not work on all aspects of a system, he needs to acquire skills to:

- Understand how one section of code fits into a larger framework
- Learn to work well with other members of a team

There might be components that work in back ground (facilitators) that needs to be taken into account

Other aspects include

- documenting the necessary pre conditions a software component requires to complete a task
- verifying that the software will perform correctly when presented with legal input values

2.11 Integration of Components:

After software subsystems have been individually designed and tested they can be integrated into the final product

"*If all modules work individually should we doubt that they will work then we put them together?*"

Problems that arise during integration of components:

- Data can be lost across an interface
- One module can have an inadvertent effect on another
- Sub functions when combined may not produce desired result
- Global data structures can present problems

Starting from a simple base elements are slowly added to the system and tested using stubs (simple dummy with no or very limited behavior)

Next, one or the other stubs can be replaced by more complete code and tested to see if the system is working as desired, this is called *Integration Testing*

The application is finally complete when all stubs have been replaced with working components

Less coupling facilitates testing components in isolation.

Errors might creep in during integration that requires changes to come of the components

This follows retesting of the components in isolation before attempt to reintegrate the software once more

Re-executing previously developed test cases following a change to a software component is called *Regression Testing*

Example:

In case of IIKH, we start integration with the *Greeter* component

To test *Greeter* in isolation stubs are written for the *Recipe Database* and the daily *Meal Plan* manager

These stubs need not do any more than print an informative message and return

With this the development team can test various aspects of the *Greeter* system (*e.g. whether button press provides correct response or not*)

Testing of an individual component is called *Unit Testing*

The team then might decide to replace the stub for the *Recipe Database* component with the actual code maintaining the stub for other portion.

2.12 Maintenance and Evolution:

After the delivery of the working version of an application comes the phase of Software Maintenance

Activities:

- Bugs must be corrected in the form of patches to existing releases or in subsequent releases
- Change in requirements due to government regulations, standardization among similar products
- Hardware change (*input/output technology*) e.g. pen based to touch panel, text to GUI based system
- Change in user expectations (*greater functionality, lower cost, easier user*) , competition
- Request for better documentation by users

A good design recognizes the inevitability of changes and plans an accommodation for them from the very beginning.

Chapter 3

3 Class, Methods and Messages

3.1 Encapsulation

For solving problems, programmers often make use of abstract data types, in which **information is consciously hidden** in a small part of a program

Each of these has two faces

From outside a client (*user*) of an abstract data type sees only a **collection of operations that define the behavior** of abstraction (*interfaces*)

On the other side of the interface, there exists **data variables that are used to maintain the internal state of the object** as defined by the programmer

Instance means a *representative*, example of a class e.g. an object is an instance of a class

Instance variable means *internal variable* maintained by an instance (for maintaining it's state)

Each instance has its own collection of **instance variables**

These values can only be changed by **methods** associated with the class and not directly by clients

An object is composed of *state* and *behavior*

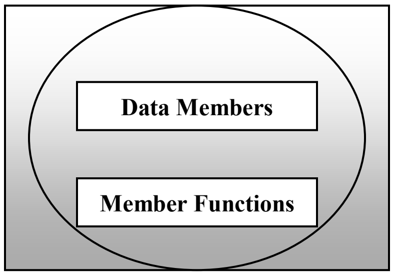
State is described by the instance variables

Behavior:

Characterized by the methods

Used for modifying state

Used for interacting with other objects



The *binding* of data and functions together into a single class-type variable is referred to as *encapsulation*

The principle of hiding the used data structure and to only provide a well-defined interface is known as *encapsulation*.

3.2 Varieties of Classes

Based on different forms of responsibility classes can be used for many different purposes and be categorized as follows:

a. *Data Manager (Data or State classes)*

Main responsibility is to maintain data or state information

Often recognizable as nouns in problem description

e.g. Card class maintains data values describing rank and suit

b. *Data Sink or Data Source*

Responsible for generating data

e.g. random number generator

Responsible for accepting data and further processing them

e.g. performing output to a disk or file

These types of classes don't hold data for any period of time

Instead, they generate data on demand (from a data source) and process it when called upon (for a data sink)

c. *View or Observer classes*

Responsible for displaying information on screen

As code for performing this activity is often complex, frequently modified and largely independent of the actual data being displayed, the display behavior is isolated from others

Separation of object being viewed (model) from the view that displays the visual representation of object
The model neither requires nor contains any information about the view

Facilitates code reuse

e.g. financial information (model) could be displayed as bar charts , pie charts or tables (views)

d. *Facilitator or Helper Class*

Responsible for assisting in the execution of complex tasks

Maintains little or no state information themselves

e.g. while displaying a playing card on screen we use services of facilitator class that handles the drawing of lined and text on screen

This list is not exhaustive

Some might not fit into any group described above

Some might span in two or more of the categories, in which case it can be split into two or more classes

3.3 Example Playing Card:

A card class is a data manager that holds and returns the rank and suit values and draws itself

Can be incorporated in any type of game (*Reusability*)

A CRC card is used for describing the behavior of a card in terms of its responsibilities.

Names and arguments list for each method can be identified

Data values to be maintained by each instance of the playing card can be identified

Finally the behavior and state described on the front and back of the CRC card is translated into *executable code*

Card

Maintain suit and rank
Return color
Maintain face-up or face-down status
Draw card on playing surface
Erase card on playing surface

A CRC card for the class Card

Card

suit – **return card suit**
rank – **return card rank**
color – **return card color**
erase, draw –**erase or draw card image**
face-up, flip – **test or flip card**

Revised version of CRC Card

3.4 Interface and Implementation:

Parnas's Principle in terms of objects:

A class definition must provide the intended user with all the information necessary to manipulate as instance of the class correctly and nothing more

A method must be provided with all the information necessary to carry out its given responsibilities and nothing more

This principle divides the world of an object into two spheres:

Interface:

The world as seen by the user of an object (the user of the services provided by an object)

Describes how the object is interfaced to the world

Implementation:

A view from within the object

The user of an object is permitted to access no more than what is described in the interface

Describes how the responsibility promised by the interface is achieved (implementation aspect)

3.5 Classes and Methods in C++

3.5.1 Structures in C and C++

Structure in C++ has expanded the capability further to suit its OOP philosophy

In C++ , a structure can have both **variables** and **functions** as members

Provides facility to hide the data, by allowing declaration of some of its members as private so that they can not be accessed directly by external functions

Supports the mechanism by which one type can inherit characteristics from other types (*inheritance*)

In C++, the key word *struct* can be omitted in the declaration of structure variable, it's an error in 'C'

C++ incorporates all these extensions in another user defined type known as Class

There's a very little syntactical difference between structures and classes in C++

The only difference between a structure and a class in C++ is that **by default the members of a class are private while by default the members of a structure are public**

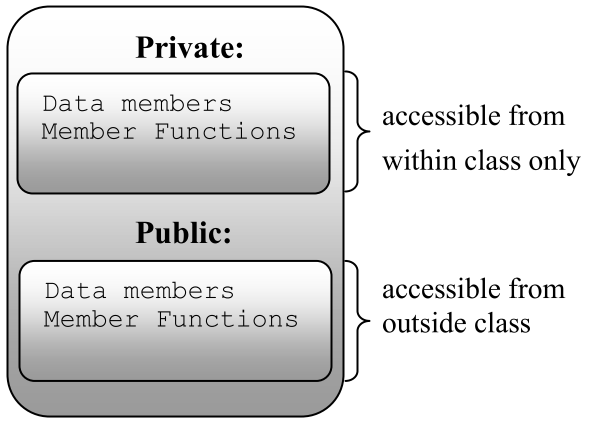
A class is an abstract data type that combines related data and functions together, which can be used to create objects (*instances/variables*) of this type.

Class descriptions begin with the key word *class*

The keyword *private* precedes those portions of code that can be accessed only by the methods in the class itself

The keyword *public* indicates the true interface - those elements that are accessible outside the class

```
class name_of_class{
 private:   variable declaration;  //Member Data
            function declaration;  //Member Function
 public:    variable declaration;
             function declaration;
 protected: variable declaration;
            function declaration;

};
```



Class

Interface files (*usually given a ".h" extension*) are kept separate from implementation files

Interface files contain description of one or more classes.

```
class student {
   private:
        char name[30];
        int roll;
   public:
        void init_data();
        void display_data();
};
```
An implementation file for a class must provide definitions for the methods described in the interface file

If the above file is saved as "*student.h*" then we include this file in the implementation file as:

```
# include "student.h"
void student :: init_data(){
      cout << " Enter name:";
      cin  >> name;
      cout << "\n Enter roll:";
      cin  >> roll;
}
```

```
void student :: display_data(){
      cout << " Name:" << name;
      cout << "\n Roll:" << roll;
}
```

The body of a function is written as a conventional 'C' function

The class name and two colons (:: *scope resolution operator*) precede the method name

The *instance variables* (*data members*) can be referenced within a method as variables

3.5.2 Inline Functions:

Appear same as other regular functions

Compiler expands *inline functions* into code directly at the point of call avoiding the overhead of function call and return instructions

Should be used only with functions with small function body

Making Inline functions:

1. The key word '*inline*' is used in the beginning of the function definition to make it inline

```
# include "student.h"
inline void student :: init_data(){
      cout << " Enter name:";
      cin  >> name;
      cout << "\n Enter roll:";
      cin  >> roll;
}
```

```
inline void student :: display_data(){
      cout << " Name:" << name;
      cout << "\n Roll:" << roll;
}
```

2. If a body of a member function is defined directly inside the class then it behaves like an inline Function. No need to use the keyword '*inline'* explicitly

   Not a good practice as it makes the class definition more difficult to read

   Should only be used when there are few methods and the method bodies are very short

```
class student {
   private:
        char name[30];
        int roll;
   public:
        void init_data();
        void display_data(){
            cout << " Name:" << name;
            cout << "\n Roll:" << roll;
        }
};
```

3.6 Message-Passing Formalism

Creation: The allocation of memory space for new object and the binding of that space to a name

Initialization : Setting of initial values in the data area for the object , similar to the initialization of fields in a record and also establishing the initial conditions necessary for the manipulation of an object

Message Passing (*Method Lookup*): The dynamic process of asking an object to perform a specific action.

How *message passing* differs from a *function call:*

i. A message is always give to some object called the receiver

ii. The action performed in response to the message is not fixed but may differ depending upon the class of the receiver. i.e. different objects may accept the same message and yet perform different action

Three identifiable parts of a message passing expression

i. *Receiver* (The object to which the message is sent)

ii. *Message Selector* (The text that indicates the particular message being sent)

iii. *Arguments*

3.6.1 Message Passing Syntax in C++

In C++ a method is described as a member function and passing a message to an object is referred to as *invoking a function*

The syntax used to invoke a member function is similar to that used to access data members

The notation states the *receiver*, followed by a *dot/period* then the *message selector* (*the name of a member in the class of the receiver*) and finally the *arguments* inside the braces.

```
syntax:

      receiver.message_selector (arguments);
```

```
class student {
   private:
       char name[30];
       int roll;
   public:
       void init_data();
       void display_data();
};
void student :: init_data(){
      cout << “ Enter name:”;
      cin  >> name;
      cout << “\n Enter roll:”;
      cin  >> roll;
}


void student :: display_data(){
      cout << “ Name:” << name;
      cout << “\n Roll:” << roll;
}
void main(){
      student s1;
      s1.init_data();
      s1.display_data();
}
```

3.7 Issues in Creation and Initialization; Stack Versus Heap Storage Allocation

Issues concern how storage for variables is allocated and released and the steps the programmer must take in these processes

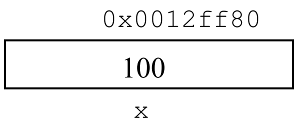
Automatic variables are different from *dynamic variables*

Automatic

Storage for an automatic variable is created when the procedure or function containing the variable is entered and released automatically when the procedure or function is exited

When the space is created for an automatic variable the name and the created space are linked and they cannot change during the time the variable is in existence

```
int x = 100;
```



Dynamic

In case of Pascal, a variable is declared as a pointer by passing the name of the variable as an argument in a procedure `new(x);` for creating a dynamic variable.

Here, space is allocated and the value of variable points to this new space

Here, the process of allocation and naming are tied together

In other languages like C, allocation and naming are not tied together

In C we used the function *malloc* that takes the amount of memory to allocate as the argument

The *malloc* call returns a pointer to a block of memory which is assigned to a pointer variable

```
struct student{
  int roll;
  char name[20];
};
```

```
void main(){
   student *s;
   s = (struct student *) malloc (sizeof (struct student));
}
```

In *C++*, we use the *new* operator

```
#include <iostream.h>
#include <conio.h>
class student{
 private:
  char name[20];
  int roll;
 public:
   void showName(){
    cout << "Name: " << name;
   }
```

```
 void setName(char n[]){
    strcpy(name,n);
 }
};
void main(){
  student *s;
  s = new student();
  s->setName("Larry");
  s->showName();
}
```

3.7.1 Essential Difference between Stack Based and Heap Based Storage Allocation

Stack based storage allocation (automatic variable):

Storage (*memory*) is allocated for an automatic (*stack-resident*) variable without any explicit directive from the user

The programmer almost never needs to consider the release of *automatic variables*

Heap based storage allocation (dynamic variables):

Storage (*memory*) is allocated for dynamic variable only upon explicit request

Programmer needs to consider the release of *heap based storage*

3.7.2 Memory recovery

In case of heap based storage allocation, some means must be provided to recover storage that is no longer being used

In case of *C, C++* and Pascal the user is required to keep track of values and free the space occupied by variables once they are no longer useful

The built in routine used for the purpose are *dispose* in Pascal, *free* in C, *delete* in C++

```
#include <iostream.h>
#include <conio.h>
class student{
 private:
  char name[20];
  int roll;
 public:
   void showName(){
    cout << "Name: " << name;
   }
```

```
 void setName(char n[]){
    strcpy(name,n);
 }
};
void main(){
  student *s;
  s = new student();
  s->setName("Larry");
  s->showName();
delete s;
}
```

Errors :

Attempt to use a memory area that has not yet been allocated

Dynamically allocated memory is never released (*memory leak*)

Attempt to use memory that has been already freed

Same memory location is being freed twice by two different procedures

Remedy:

Ensuring that every dynamically allocated memory object has a designated owner and is responsible for freeing memory when no longer in use

In Java and Smalltalk when values are no longer used they are automatically detected, collected and their space is recovered and recycled for future usage

This process is known as *garbage collection*

Automatic memory recovery (*garbage collection*) can be expensive and necessitates a run time system to manage memory (*background process*)

3.7.3 Pointers;

Pointer are efficient and effective means of dealing with dynamic information

In some languages (*Java, Smalltalk and object Pascal*) objects are represented internally as pointers but are not used as pointer by the programmers

In other languages (*C++*) the user must explicitly distinguish between variables holding values and variables holding pointer to values

3.7.4 Immutable Creation

For some objects, the values associated with the variable are set only once and not altered there after during the course of execution these are known as single assignment or immutable

An object for which all instance variables are immutable is known as an immutable object
e.g. in the class Card, the instance variables suit, rank are set only once and don't change

Immutable Values

- Can be assigned only once
- The value is not determined until execution time when the object containing the value is created

Program Constants

- Must be known at compile time
- Has global scope
- Remains fixed

Static member variables

Static member variables are **used to maintain values common to the entire class** and **shared by all objects of that class**.

Properties:

- Static member variable of a class is **initialized to zero** when the first object of its class is created.
- **Only one copy** of that member is created for the entire class and is shared by all the objects of that class
- It is **visible only within the class** but its **life time is the entire program**

The **type** and **scope** of a static member variable **must be defined outside the class definition** (static data members are stored separately and not as a part of an object). They are also called **class variables** as they are not only associated with just one class object.

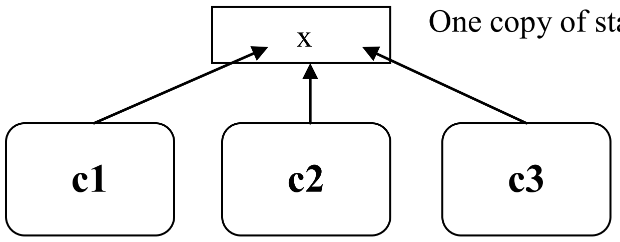
Syntax:

```
data_type class_name :: static_member_variable_name  //definition of static data member

class count {
    static int x;      // declaration of static data member
 public:
  void getCount(){
     x++;
     cout<<"\n x is "<< x;
  }
};
 int count :: x;       //definition of static data member

 void main(){
  count c1;        //creation of first object of class count
  c1.getCount();

  count c2;            //creation of second object of class count
  c2.getCount();

  count c3;            //creation of third object of class count
  c3.getCount();
 }
```

What will be the output of the program?

```
  x is 1
  x is 2
  x is 3
```

What would be the output of the program if we hadn't declared x as static?



One copy of static data member x is shared by objects c1,c2 and c3

While defining a static variable some **initial value can also be assigned to the variable**

For example in the above example,

```
int count :: x = 99;       //static data member initialized to 99
```

Static member functions

Static member functions is a member function of a class that **can access only the static data** member of the class
The keyword **static** is used to precede the member function to make a member function static
The static member function **acts as global for members of its class** without affecting the rest of the program
The **purpose of static member is to reduce the need for global variables** by providing alternatives that are local to a class
A static member function **is not part of objects of a class**
The **keyword** static should **be used only in the declaration** part on the function definition part

```
class count {
    static int x;          // declaration of static data member
 public:
  count(){
    x++;
  }
  static void getCount(); // declaration of static function
};
 int count :: x;           //definition of static data member
 void count::getCount()    // definition of static function
 {
     cout<<"\n x is "<< x;
 }
void main(){
  count::getCount();          //before creation of objects of class count
  count c1,c2,c3,c4;       //after creation of object of class count
  count::getCount();
  getch();
}
```

3.8 Mechanisms for Creation and Initialization in C++

3.8.1 Constructors

Implicit initialization is facilitated in `C++` through the use of a special member function called *constructors*

A *constructor* is a method with the **same name** as that of the object class

A *constructor* method is **automatically** and **implicitly invoked** any time an object of the associated class is created

- During variable declaration (*static creation*)
- During creation of objects *dynamically* with the new operator etc

Constructors **don't have return types** (*not even void*) in contrast with normal functions

They are declared in the **public** section of a class

A constructor **without arguments** is known as *default constructor*

```
class integer{
int m,n;
public:
   integer(void);
   void showInteger(){
   cout << "m:"  << m << " n:" << n;
}
     -----------
};
```

```
integer :: integer(void){
  m = 0; n = 0;
};
void main(){
  integer i;
  i.showInteger();
}
```

Arguments can be passed to the constructor function to initialize various data elements of different objects with different values when they are created.

These constructors that **can take arguments** are called *parameterized constructors*

```
class integer{
int m,n;
public:
   integer(int a, int b);
   //parameterized constructor
   void showInteger(){
   cout << "m:"  << m << " n:" << n;
}
-----------
};
```

```
integer :: integer(int a, int b){
   m = a;
   n = b;
};
void main(){
 //  integer i; will not work
 integer i(1,2); //implicit call to constructor
 //integer i = integer(1,2); explicit call
  i.showInteger();
}
```

A function name is said to be *overloaded* when there are two or more function bodies known by the same name and they are disambiguated by difference in parameter lists

This capability is used by constructors to **provide more than one style of initialization** called *overloaded constructors*

```
class integer{
int m,n;
public:
      integer();
      integer(int a);
      integer(int a, int b);
      void showInteger(){
      cout << "m:"  << m << " n:" <<n;
}
};
integer :: integer(){
      m = 10; n =1 0;
};
```

```
integer :: integer(int a){
      m = n = a;
};
integer :: integer(int a, int b){
      m = a; n = b;
};
void main(){
  //  integer i;   invokes first constructor
  //integer i(1);  invokes second constructor
  integer i(1,2);  // invokes third one
  i.showInteger();
}
```

If there are more than one constructor functions associated with the class, the one invoked depends upon the arguments used when value is created.

The body of a constructor is simply a sequence of assignment statements

These statements can be replaced by *initializers* in the function heading

Each *initializers* clause names an instance variable and in parentheses lists the value to be used to initialize the variable.

```
class integer{
int m,n;
public:
  integer(int a, int b);
  void showInteger(){
    cout << "m:"  << m << " n:" <<n;
  }
};
```

```
integer :: integer(int a, int b): m(a), n(b)
{/*no further initialization necessary*/}

void main(){
 integer i(1,2); //implicit call to constructor
  i.showInteger();
}
```

New

In C++, dynamic values are created with the key word **new** followed by the name of the object and arguments to be used with the constructor for the object

The result is a pointer to a newly created object.

```
class integer{
int m,n;
public:
      integer();
      integer(int a, int b);
      void showInteger(){
       cout << "m:"  << m << " n:" <<n;
      }
};
integer::integer(){
  m = 1; n = 2;
}
```

```
integer::integer(int a, int b){
  m = a; n = b;
}
void main(){
  //integer *i;
  //i = new integer(1,2);
      // or
  integer *i = new integer;
  i->showInteger();
}
```

An array of values can be created by a size give in square brackets

The size argument is then computed at run time

```
integer * i = new integer[20];
```

Memory for dynamically allocated values must explicitly released by the programmer using delete for an array of objects

Values can be declared as **immutable** in C++ thru the use of ***const*** keyword, such values are declared as constant and are not allowed to change, the instance variables that are constant must be *initialized*

3.8.2 Overloaded Assignment Operator and Copy Constructor

```
#include <iostream.h>
#include <conio.h>
class code{
  int id;
  public:
   code (){}            //constructor
   code(int a){ id = a;} //constructor
   code(code & x){ //copy constructor
     id = x.id;
   }
   void display(){
    cout << id;
   }
};
```

```
void main(){
 code A(100); //A created and initialized
 code B(A);   //copy constructor called
 code C = A;  //called again

 code D; //D created but not initialized
 D = A;  //copy constructor not called

 cout << "\n id of A: " ; A.display();
 cout << "\n id of B: " ; B.display();
 cout << "\n id of C: " ; C.display();
 cout << "\n id of D: " ; D.display();
}
```

Assignment operator:

In the above example,

```
code D;
D = A;
```

`D` and `A` are objects of the type code
The statement `D = A;` causes the compiler to copy the data from `A` member by member into `D` this is what the assignment operator does by default

Copy Constructor

Used for declaring and initializing an object form another object.

```
code  B(A);
or circle C = A;
```

Here, we have initialized one object with another object during declaration

The compiler creates a new objects `B,C` and copies the data from `A` member by member into `C and B`

This is what the *copy constructor* does by default

Difference between copy constructor and assignment operator:

Copy constructor also **creates a new object** apart from initialization

The *assignment operator* **does not** create new object.

3.8.3 Destructors

A ***destructor*** is a **function** that is **called automatically whenever an object is destroyed** (*control goes outside main()*)

A *destructor* **has the same name as the constructor (*or class*)** but is **preceded by a tilde ~**

A *destructor* **does not have a *return* value**

A *destructor* **does not take any arguments** (*the assumption being that there are many ways to construct an object but there's only one way to destroy an object*)
The prime use of destructors is to ***de-allocate*** **memory** that was allocated for the object by the *constructor*

```
#include <iostream.h>
#include <conio.h>
int count=0;

class trace{
public:
trace(){
  count++;
  cout << "\nCreated object no[" << count << "]";
}
~trace(){
  cout << "\nDestroyed object no[" << count << "]";
  count--;
  getch();
 }
};
```

```
void main(){
 cout << "\n In Main:";
 trace t1,t2,t3,t4;
 {
   cout << "\n\n  Entering Block One ";
   trace t5;
 }
 {
   cout << "\n\n  Entering Block Two ";
   trace t6;
 }
 cout << "\n\n Re-entering Main:";

 getch();
}
```

This Pointer

Chapter 4

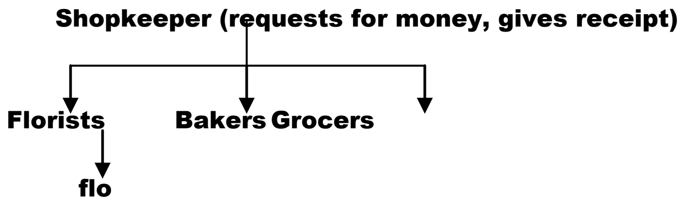
4. Inheritance

Inheritance is the property that instances (objects) of a child class (*sub class*) can access both data and behavior (methods) associated with a parent class (*super class*)

The mechanism of creating a new class (*derived class*) from an old one (*base class*) is called *inheritance* or *derivation*.

The derived class inherits some or all the capabilities of the base class and **can also add new features and refinements of its own**.

Inheritance provides the concept of ***code reusability***. (*A programmer can use a class created by another person without modifying, derive other classes from it that are suited for a particular programming situation*)

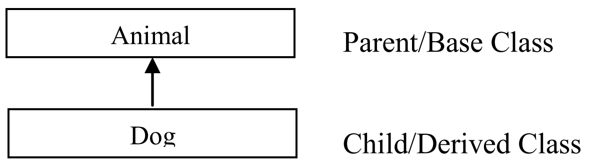


A **child** class is a **specialized** (*restricted*) form of the parent class so it is a ***contraction*** of the parent class in the mean time child class are also an ***extension*** of the parent as they extend the properties associated with parent class

Inheritance is always ***transitive*** so that a class can inherit features from super classes many levels away
e.g. Dog is a subclass of Mammal and Mammal is a subclass of Animal so a dog will inherit attributes both from Mammal and Animal

The *symbolic* representation of inheritance is ↑

The **direction of the arrow** (*arrow-head*) **points towards the parent class** meaning that the derived class *refers* to the functions and data in the *base class* while the base class has no access to the *derived class* data or functions



It establishes **"is *a kind of*"** relationship e.g. *a dog is a kind of animal*

4.1 Subclass, Subtype and Substitutability

The relationship of the data type associated with a parent class to the data type associated with a child class gives rise to the following arguments:

i. Instances of the subclass must possess all data areas associated with parent class
ii. Instances of the subclass must implement thru inheritance at least all functionality defined for the parent class (*They can also define new functionality*)
iii. An instance of a child class can mimic the behavior of the parent class and should be indistinguishable from an instance of the parent class if substituted in a similar situation

The Principal of Substitutability
"If we have two classes A and B such that class B is a subclass of A, it should be possible to substitute instances of class B for instances of class A in any situation with no observable effect"
Subtype is subclass relationship in which *the principle of substitutability* is maintained however other forms of subclass may not satisfy this principle.

In the example below the class B object is replaced by the object of class A without any error. This is achieved when a child class is inherited from a base class **publicly.** In the same example, if the mode of inheritance is made **private** the base class object 'a' can't be replaced by the parent class object 'a'

```
class A
{
  public:
   void printMsg(){
      cout <<" I am A!!";
   }
};
class B : public A
{ public:
   void printMsg(){
    cout <<" I am B!!";
   }
};
```

```
void testingA(A  a)
{
   a.printMsg();
}
void testingFunc()
{
   B b;
   cout<<" Passing B"<<endl;
   testingA(b);
         // b substituted for object of A.
}
void main(){
   testingFunc();
}
```

4.2 Forms of Inheritance

Description of categorization of the uses of inheritance

Two or more descriptions may be applicable to a single (same) situation

a. Subclassing for Specialization (Subtyping)

The most common use and ideal form of inheritance

The new class is a specialized form of the parent class but satisfies the specifications of the parent in all relevant respects

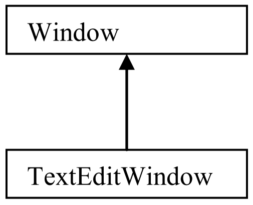
The principle of substitutability is explicitly upheld

A *Window* class provides general windowing operations like moving, resizing

A specialized subclass *TextEditWindow* inherits the window operations

In addition it provides facilities that allow the window to display and edit textual material

TextEditWindow satisfies all the properties of a window thus is a subtype of window in addition of being a subclass



b. Subclassing for Specification

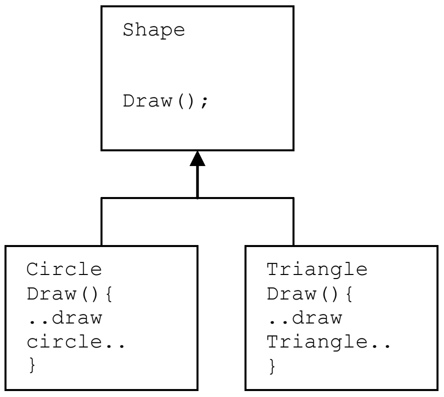
Here the classes maintain a certain *common interface* (*they implement the same methods)*

The parent class contains combination of *implemented operations* and *operations that are deferred to the child class*

There is *no interface change* as the child implements the behavior described but not implemented in the parent

This is a special case of *subclassing for specialization* where subclasses are realization of an *incomplete abstract specification* and they are not refinements of an existing type

The parent class is called *abstract specification* class as it does not implement actual behavior but only defines them



c. Subclassing for Construction

Here , a class often inherit almost all of its desired functionality from a parent class changing only the names of the methods used to interface to the class or modifying the arguments in certain fashion
e.g. classes are created to write values to a binary file

Parent class implements only the ability to write raw binary data

A subclass is constructed for every structure that is saved using the behavior of the parent class

```
class storable{
 void writeByte(unsigned char);
};
class storeStruct: public storable{
 void writeByte(myStruct &aStruct);
};
```

d. Subclassing for Generalization

Opposite of *Subclassing for specialization*

Subclass extends the behavior of parent class to create more general kind of object

Often applicable when we build on a base of existing classes that we do not wish to or can not modify
e.g. a class Window is defined for displaying on a simple back and white background

now we can create a subtype *ColoredWindow* that lets the background color to be something else by adding additional field to store the color and overriding the inherited behavior of parent window

Occurs when the overall design is based *primarily on data values* and only *secondarily on behavior*
e.g. in previous case colored window contains data fields that are not necessary in the simple window case

e. Subclassing for Extension

Subclassing for generalization modifies or expands on the existing functionality of an object where as *subclassing for extension* adds totally new abilities

In *subclassing for generalization*, at least one method from the parent must be *overridden* and the functionality is tied to that of the parent

Extension simply adds new methods to those of the parent and the functionality is less strongly tied to the existing methods of the parent

e.g. a *StringSet* that is specialized for holding string values whose parent class is *Set* class
StringSet class may provide additional string related methods that are not relevant to the parent class

e.g. "*search_ by_ prefix()*" method that returns a subset of all the elements of the set that begin with certain value

f. Subclassing for Limitation

Occurs when the behavior of the subclass is smaller or more *restrictive* than the behavior of the parent class

Used when a programmer cannot or should not modify the parent class
e.g. in case of a *deque* double ended queue, elements can be added or removed form either end of it

The *deque* can be converted into a *stack* (in which elements can be added or removed from only one end) by eliminating the functionality of *deque*

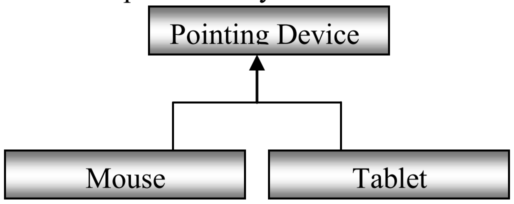
It doesn't follow the rule of substitutability as it builds subclasses that are not subtypes so it should be avoided

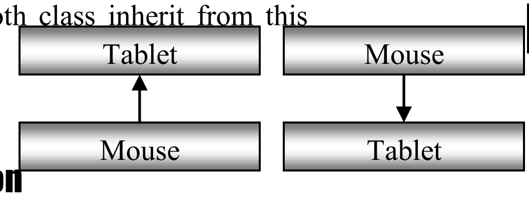
g. Subclassing for Variance

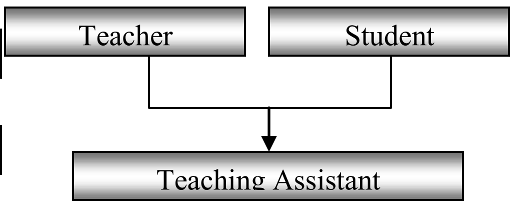
Used when two or more classes have similar implementations but do not seem to possess any hierarchical relationships between concepts represented by the classes

e.g. code required to control a *mouse* and a *tablet* are identical

In this case one of the two classes can be chosen as the parent by inheriting the common code and overriding the device specific code A better alternative, factor out the common code into a abstract class, *PointDevice* and have both class inherit from this common parent class







h. Subclassing for Combinatio

Here , the subclass represents a combination of features from two or more parent classese.g. a teaching assistant have characteristics of both a teacher and a student
The ability of a class to inherit from two or more parent classes is known as *multiple inheritance*

4.3 Inheritance- Its merit and demerits

4.3.1 Benefits of Inheritance (Merits)

i. Software Reusability

- The child class need not rewrite the behavior inherited from its parent class
- e.g. functions for inserting a new element into a table, to search for a pattern in a string can be written once and reused
- Other benefits of reusable code include increased reliability (opportunities for discovering errors)

ii. Code Sharing

- many user or projects can use the same class (software ICs)
- code sharing thru inheritance

iii. Consistency of Interface

- All child classes inherit the same behavior from their parent class
- This means that interfaces to similar objects are similar which means the user is not presented with a confusing collection of similar objects that are interacted with and behave differently

iv. Software Components

- Inheritance facilitates building of reusable software components
- The aim is to allow development of applications that require little or no actual coding thru the use of ready made software components / libraries (e.g. Java Beans)

v. Rapid Prototyping

- usage of reusable components cuts down time required for coding which can be spent on understanding the new and unusual portions of the system
- Thus software can generated quickly and early leading to a style of programming called Rapid Prototyping or exploratory programming
- Here, a prototype system is developed user experiments with it, a second system is produced that is based on the experience with the first and so on for several iterations
- Useful where the requirements of the system are vaguely understood when the project begins

vi. Polymorphism and Framework

- In case of conventional programming (e.g. structured) software is written from the bottom up although they may be designed from the top down
- Lower level routines are written on top of these slightly higher abstractions are produced and so on
- Code portability decreases as one moves up the levels of abstractions
- Lower level pieces are independent and portable
- But due to declarations and data dependencies the higher level components are tied to a particular application making them less portable
- In OOP, Polymorphism allows the programmer to generate high-level reusable components that can be tailored to fit different applications by changes in their lower parts

vii. Information Hiding

- Programmer reusing a software component needs only to understand the nature of thecomponent and its interface
- Implementation details of the components doesn't concern him

- This reduces interconnectedness between software systems
- Interconnected nature of conventional software is one of the principle cause of software complexity

4.3.2 Costs of Inheritance (Demerits)

i. Execution Speed

- Inherited methods are often slower as they must deal with arbitrary subclasses
- But decrease in execution speed may be balanced by an increase in the spped of software development

ii. Program Size

- Making use of software library increases the size of project
- But as memory costs decrease the size of program becomes less important
- Focus is more on producing high quality and error free code rapidly

iii. Message Passing Overhead

- Message passing is more costly operation that simple function call (requires additional assembly language instructions)
- To over come this , member functions are invoked by a class name in C++ and inline functions expanded

iv. Program Complexity

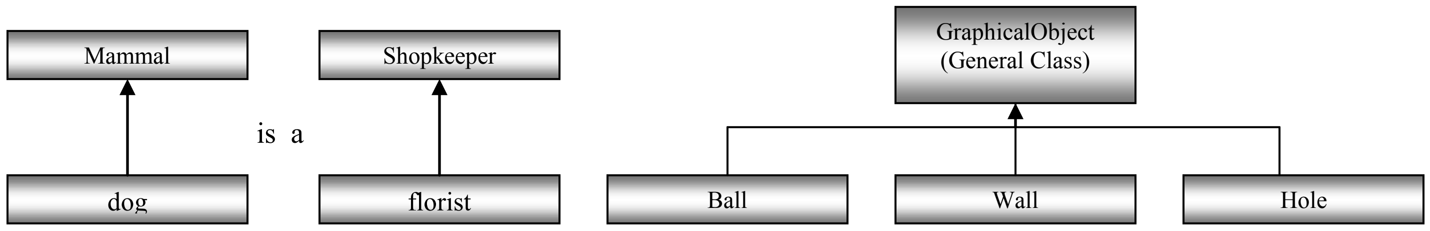
- Overuse of inheritance can replace one form of complexity with another
- i.e. to understand the control flow of a program a program may require to scan up and down the inheritance graph several times
- This is known as the *yo-yo* problem

4.4 Inheritance and Substitutability

Relation between a variable declared as one class and value derived from another class.

It should be legal to assign the value to the variable if the class of the value is the same as or subclass of the class of the variable

In order to know if an instance of some class can be substituted with another one we can use the "*is-a* "rule



4.4.1 The "*is-a*" rule and the "*has-a*' rule

Knowing the difference between the two helps in knowing how and when to apply object oriented software reuse techniques

4.4.2 Is-a relationship

If the first concept is a specialized instance of the second one then there exists an "*is-a"* relationship between them

The behavior and data associated with the more specific idea form a subset of the behavior and data associated with the more abstract idea

Testing the relationship:

X is a specialized instance of concept *Y* if the assertion in plain English "*X is a Y*" sounds correct or logical

Examples of inheritance satisfying the "*is-a"* relationship *a florist is a shopkeeper*, *a dog is a mammal*

4.4.3 Has-a relationship

If the second concept is a component of the first and the two are not in any sense the same thing

e.g. *a car has a engine* here, neither *a car is a engine* nor *an engine is a car* but *a car is a vehicle*

Testing the relationship:

Form a simple sentence "*an X has a Y*", if the result sounds reasonable then the relationship holds

4.5 Composition and Inheritance

4.5.1 Inheritance:

In case of inheritance, apart from its own, the derived class acquires all the characteristics from the base exhibiting a "**is a**" relationship with the parent.

4.5.2 Composition:

In case of composition, a class or concept incorporates another class which bears no similarity among themselves thus exhibiting a "**has a**" relationship.

We can say that one object is composed of other objects, to make it a whole. e.g *a car has an engine*

Also called a "**part of**" relationship e.g. *an engine is a part of car*

Here, since one object is **composed of** or **contained in** another one, this relationship is also called **containership**

```
class engine{

};
```

```
class car {
   engine e;    // a car is composed of an engine
};
```

When composition is employed to reuse an existing data abstraction in the development of a new data type , a portion of the state of the new data structure is an instance of the existing structure

The implementation of the *startEngine* operation for our car data structure invokes the similarly named function already defined for *engine class*

```
class engine{
public:
    void startEngine();
};
```

```
class car {
 engine e;
 public:
   void startEngine(){
       e.startEngine();
   }
};
```

Initialization of data members in the engine class can be done at the same time when the constructor for class car is invoked

```
class engine{
int modelNo;
 public:
   engine(int x){
      modelNo = x;
   }
 void startEngine(){
   cout<<".. started engine "<<modelNo;
 }
};
```

```
class car {
 engine e;
  public:
       //constructor invoking another one
  car():e(33431){ }

  void startEngine(){
     e.startEngine();   }
};
void main(){
   car c;
   c.startEngine();
}
```

4.5.3 Composition and Inheritance Contrasted

- Composition **is the simpler** of the two techniques. The **operations** that can be performed on a particular data structure **are clearly indicated**
- In inheritance, the **operations of the new data abstraction are a superset of the operations of the original data structure** on which the new object is build. To know the legal operations for the new structure the programmer must **examine the whole inheritance chain**.
- In case of inheritance, it is not necessary to write any code to access the functionality provided by the class on which new structure is built. So, **implementations using inheritance are shorter** in code than with composition.
- Inheritance does not prevent users from manipulating the new structure using methods from the parent class, even if these are not appropriate.
- In case of composition (each component performs a specific task), codes are longer but that's all a programmer needs to understand to use it but **in case of inheritance, he needs to understand the parent** (selection of the behavior) and the child class
- In case of inheritance, one additional function call (overhead) is avoided hence execution time of it is smaller than that in case of composition (but can be avoided using in line functions in C++)

4.6 Software reusability

In early days of object oriented programming it was thought that composition and inheritance would provide a way to create software from general purpose interchangeable components.

Some progress has been made but the overall process has not lived up to the early expectations:

Reasons why software reusability is difficult with Inheritance and Composition:

- Inheritance and composition provide the means for producing reusable components but they do not provide guidelines for how such a task should be performed.
- Producing useful software (*reusable*) component is difficult than developing special purpose software
- Producing reusable software component :
  - Difficult
  - Benefits cannot be realized with in a single project
  - Slows down project development

  The cost of such development must be amortized over many programming projects instead of seeking immediate benefits

  But it may not be feasible as each project usually has its own budget and schedule
- As benefits of developing reusable components do not immediately improve a project , there is little incentive for programmers to strive towards reusability
- As each new problem requires a slightly different set of behaviors, it is difficult to design a truly useful and general purpose software component the first time

  Useful reusable software components evolve slowly over many projects until they finally reach a stable state
- Lack of trust and hesitant to use software not developed "*in-house*", "*not invented here* "syndrome,
- Managers pride themselves on the quality of their programmer teams
- Many programmers have little formal training, or have not kept pace with recent programming innovations, lack of awareness regarding the availability for developing reusable software components

4.7 Inheritance in C++

4.7.1 Defining Derived classes

In C++ a derived class has to identify the class from which it is derived (its base class) in addition to its own details

The syntax of defining a derived class:

```
class derived_class_name : visibility_mode base_class_name{
     ---------
     --------- // members of derived class
};
```

The colon (:) indicates that the `derived_class_name` is derived from `base_class_name`

The visibility mode may be *private, public or protected* but the default mode is *private*

The visibility mode controls the availability of thee inherited base class members in the subclass

```
class ABC{
   // members of ABC
};
class XYZ : private ABC{ //private derivation

};
class XYZ : public ABC{  //public derivation
};
class XYZ : ABC{        //private derivation
};
```

A *derived class* inherits all the members of a base class but the derived class has access privilege only to the *non-private* members of its *base class*

Functionality of the *base class* is *extended* when we add our own data and member function in the *derived class*

4.7.2 Making a Private Member Inheritable

Protected is a visibility modifier that serves a *limited purpose* in inheritance

A **member** declared as **protected** is **accessible by the member functions with in its class and any class immediately derived** from it

It **cannot be accessed** by the function outside these two classes

```
class alpha{
  private:
      //optional , visible to member functions within  its class
  protected:
      //visible to member functions of its own class and
      //immediately derived class only
   public;
      //visible to all functions in the program
};
```

4.7.3 Visibility Modes

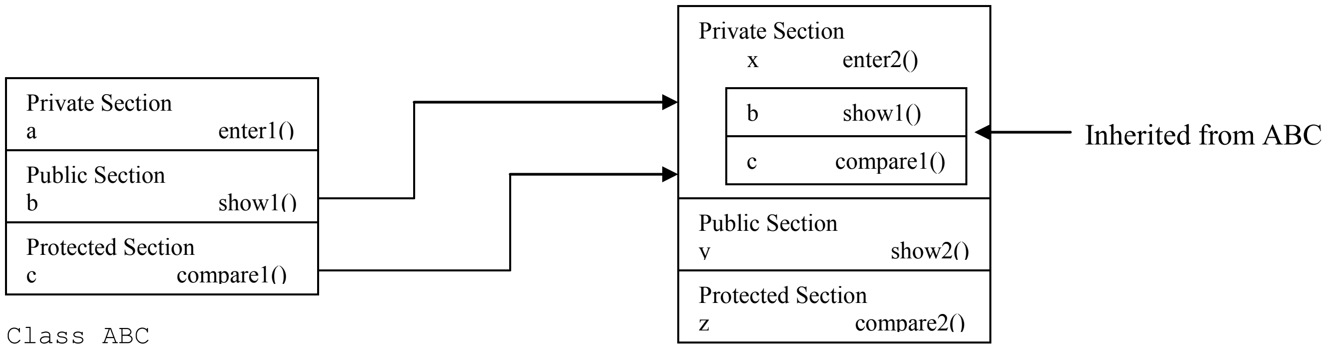
These modes control the access specifier to be inheritable members of the base class in the derived class.

i. Private visibility mode

When a *base class* is *privately* inherited by a *derived class,* the "*public members*" of the *base class* become *private members* of the *derived class*

The *public members* of the *base class* can only be accessed by the *member functions* of the *derived class*

No member of the *base class* is accessible to the *objects* of the *derived class*



```
       Class ABC
                                                          Class XYZ
class ABC{                                      class XYZ : private ABC{   //privately inherited
    int a;                                          int x;
    void enter1();                                  void enter2();
  public:                                         public:
    int b;                                          int y;
     void show1(){                                   void show2();
      cout << ".. is it working?";               protected:
     }                                               int z;
 protected:                                          void compare2();
     int c;                                     };
     void compare1();                           void main(){
};                                                   XYZ test;
                                                     test.show1();
                                                }
```

Demonstration of `private inheritance` in which the `objects of class 'y'` can not access `public member` functions of `class 'x'`.

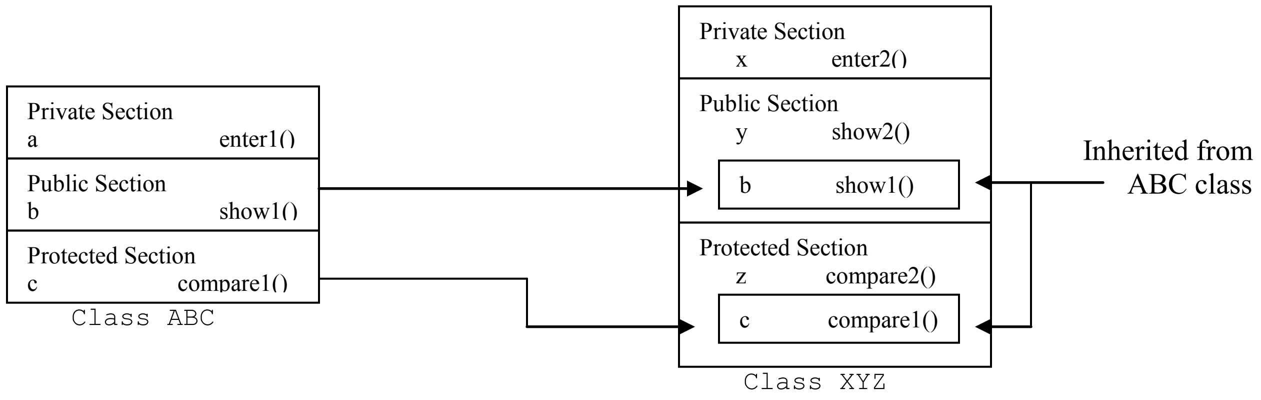
```
class x{                                                 class y : private x{
    int a;                                                   int c;
  public:                                                  public:
    int b;                                                  void mult(){
     void get_ab(){                                          get_ab();
       cout << "\nEnter values for a and b: ";               c = b * get_a();
       cin >> a >> b;                                        // a cant be used directly
     }                                                      }
     int get_a(){                                         int display(){
        return a;                                          show_a();
     }                                                      cout << " b : " << b << " c: " << c;
     void show_a(){                                       }
        cout << " a: " << a;                             };
     }                                                   int main(){
};                                                        y d;
                                                          //d.get_ab();d.show_a();d.b = 20; error
                                                            d.mult();
                                                            d.display();
                                                           d.mult();
                                                           d.display();
                                                         }
```

ii. Public Visibility Mode

Here, the *derived class* can access the *public* and *protected members* of the *base class publicly*

The *public members* of the *base class* become the *public members* of the *derived class* and become accessible to the *objects* of the *derived class*

The *protected members* of *the base class* become the *protected members* of the *derived class*



```
class ABC{
    int a;
    void enter1();
  public:
    int b;
    void show1(){
      cout << ".. its working.";
    }
 protected:
     int c;
     void compare1();
};
```

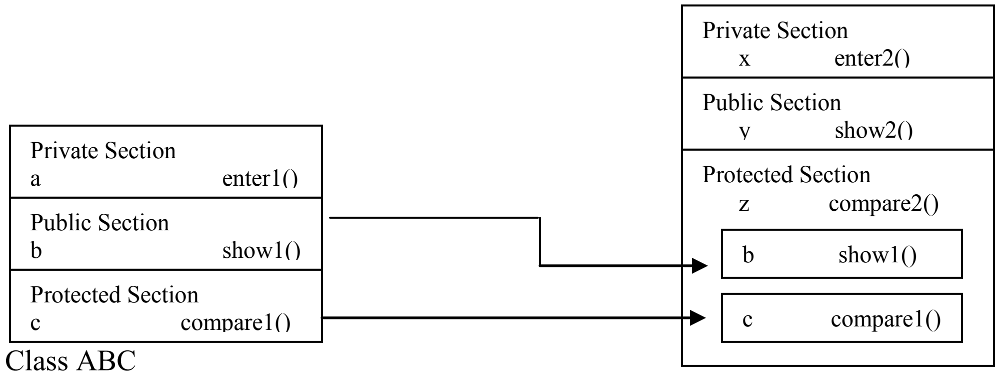
```
class  XYZ  :  public  ABC{       //publicly
inherited
    int x;
    void enter2();
  public:
    int y;
     void show2();
 protected:
     int z;
     void compare2();
};
void main(){
XYZ test;
 test.show1();   // now works
}
```

iii. Protected Visibility Mode

When the visibility mode is *protected* , the *derived class* can access the *public* and *protected members* of the *base class protectedly.*

The *public* and *protected members* of *the base class* become *protected members* of the *derived class*

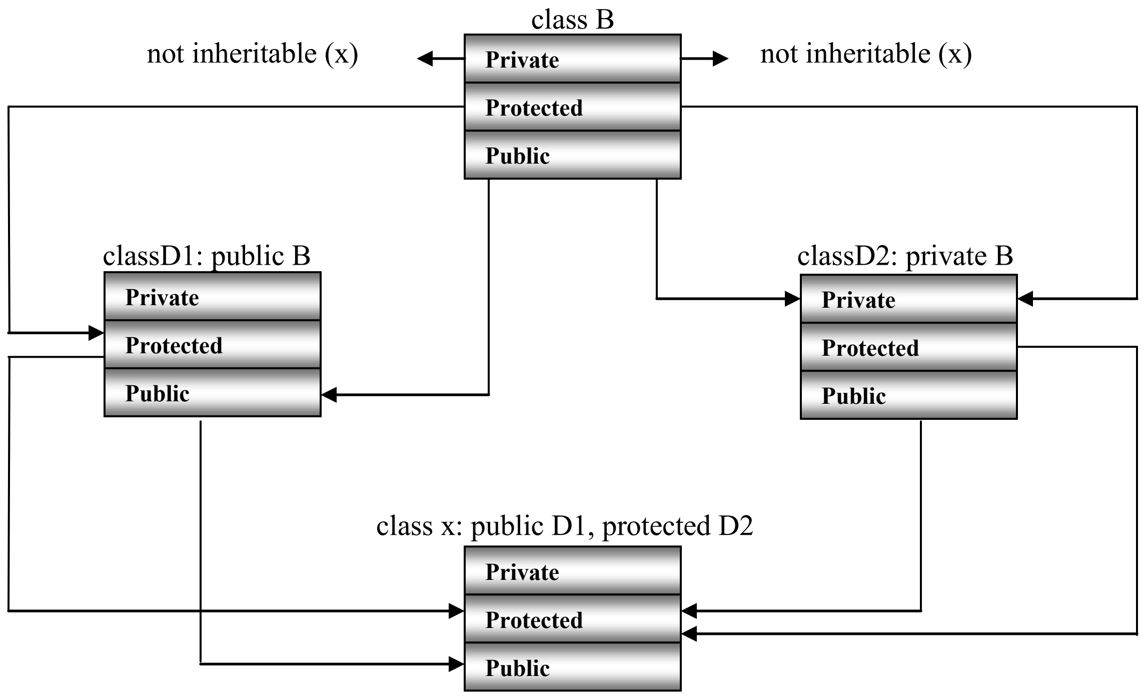
The *inherited members* are available to the *outside world* thru the *member functions* of the *derived class* and the *subclasses* of the *derived class*



Class ABC

Class XYZ

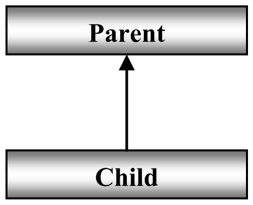
Pictorial Representation for Two Levels of Derivation



4.7.4 Types of Inheritance

4.7.4.1 Single level Inheritance

When a *derived class* inherits only from *one base class* it known as a *single level inheritance*



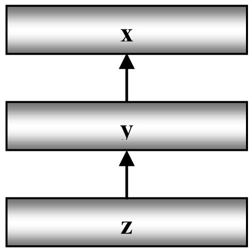
```
class x{
   int a;
   public:
    int b;
     void get_ab(){
       a =5;  b=10;
     }
     int get_a(){
        return a;
     }
     void show_a(){
        cout << " a: " << a;
     }
};
class y : public x{
    int c;
  public:
   void mult(){

    c = b * get_a();
    // a cant be used directly
   }
 int display(){
   cout << "a: " << get_a();
   cout << " b : " << b << " c: " << c;
 }
};
```

```
int main(){
 y d;
 d.get_ab();d.show_a();
  d.b = 20; //no error
  d.mult();
  d.display();
  d.mult();
  d.display();
}
```

4.7.4.2 Multi-Level Inheritance

If a *derived class* itself acts as *a base class* of another one, this is called *multilevel inheritance*

Person	x	Base Class
Employee	y	Intermediate Base Class
FullTime	z	Derived Class



```
class x{
      …  //base class members
};
class y : public x{
      …  //y derived from x
};
class z: public y{
      …  //z derived from y
};
```

Here, xyz is called *an inheritance path* and can be extended further

Assume that information about the salary of an employee is stored in three different classes. Class person stores the name and age, class employee stores the designation, basic pay, rent and class calc_salary stores the total salary of the employee. The class calc_salary can inherit the details of the employee and the name and age thru multileveled inheritance.

```
class person{
  protected: //to provide access to derived classes
    char name[10];
    int age;
  public:

    void getdata1(){
      cout << "\nEnter name";
      gets(name);
      cout << "\nEnter age";
      cin >> age;
    }
    void putdata1(){
      cout << "\nName: " << name << "\nAge: " << age;
    }
};

class employee:public person{
  //first level of derivation
  protected: //to provide access to derived classes
    char desig[10];
    int basicPay, rent;
  public:
    void getdata2(){
     cout << "\nEnter designation";
     gets(desig);
     cout << "\nEnter basic pay";
     cin >> basicPay;
     cout << "\nEnter rent";
     cin >> rent;
    }
    void putdata2(){
     cout << "\nDesignation: " << desig ;
     cout << "\nBasic Salary: " << basicPay << "\n Rent: " << rent;
    }
};

class calc_salary: public employee{   //second level of derivation
  protected:
   float total_salary;
  public:
   void display() {
     total_salary = basicPay + rent;
     putdata1();
     putdata2();
     cout << "\nTotal Salary: " << total_salary;
   }
};

int main(){
   calc_salary obj;
     obj.getdata1();
     obj.getdata2();
     obj.display();
}
```

Demonstration of Multilevel Inheritance:

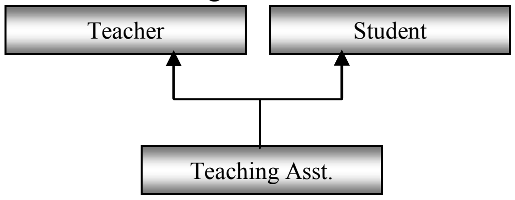
```
class student{
 protected:
  int roll;
   public:
  void getRoll(int r){
   roll =r;
  }
  void putRoll(){
   cout<<" Roll: "<<roll<<endl;
  }
};
class test:public student{
  protected:
  float sub1, sub2;
  public:
   void getMarks(float s1, float s2){
    sub1=s1; sub2=s2;
   }
   void putMarks(){
    cout<<"Marks Sub1"<<sub1<<"Sub2  "<<sub2;
    }
};
```

```
class result:public test{
   float total;
   public:
    void display(){
     total = sub1+sub2;
     cout<< " Total Marks: "<<total;
    }
};


void main(){
  result st1;
  st1.getRoll(1);
  st1.getMarks(20.3,22.3);
  cout<<"Result of :"<<endl<<endl;
  st1.putRoll();
  st1.putMarks();
  st1.display();
}
```

4.7.4..3 Multiple Inheritance

WheNBn a *subclass* inherits form *multiple base classes*, it is known as *multiple inheritance*. E.g. child inheriting the characteristics of the parents

Allows us to combine the features of several existing classes into a *derived class*



```
class derived : visibility base1, visibility base2 {
--------
-------- // body of derived class
--------
--------
};
```

```
class c : public a, public b {
--------
-------- // body of derived class
--------
--------
 };
```

Where visibility may be either *public* or *private,* the *base classes* are separated by *commas*

```
class one{
 protected:  //to  make  it  available  to
derived class
 int x;
 public:
  void disp1(){
   cout << "\n"<<x;
  }
};
class two{
 protected:  //to  make  it  available  to
derived class
 int y;
 public:
  void disp2(){
   cout << y<< "\n";
  }
};
```

```
class derived: public one, public two{
  public :
  void enter (int a, int b){
   x = a;
   y = b;
  }
};
void main(){
  int  value1, value2;
  derived obj;
  cout << "Enter two values";
  cin >> value1 >> value2;
  obj.enter (value1, value2);
  cout << " you entered";
  obj.disp1();
  obj.disp2();
}
```

Ambiguities

A lot of ambiguities can creep in when a class inherits from multiple base classes

Two classes having same function name:

```
class one{
public:
  void display(){
   cout << "\nYou are in class ONE";
  }
};
class two{
public:
  void display(){
   cout << "\nYou are in class TWO";
  }
};
```

```
class three: public one, public two{

};


void main(){
  three obj;
  obj.display();//**error**ambiguous call
}
```

Solution:

i. Function overriding: redefine the members in the derived class using the *class resolution* operator with the function, In the above program replace the class three with the one below

```
class one{
public:
  void display(){
   cout << "\nYou are in class ONE";
  }
};
class two{
public:
  void display(){
   cout << "\nYou are in class TWO";
  }
};
```

```
class three: public one, public two{
  public:
    void display(){
      one::display();  //<===
    }
};
void main(){
  three obj;
  obj.display();//**error**ambiguous call
}
```

Adding theory and practical marks to compute the Total marks of a student.

```
class Theory{
  protected:
  float tMarks;
  public:
  void gettMarks(float t){
   tMarks =t;
  }
  void display(){
   cout <<"Marks in Theory: "<< tMarks<<endl;
  }
};
class Practical{
  protected:
  float pMarks;
  public:
  void getpMarks(float p){
   pMarks =p;
  }
 void display(){
 cout <<"Marks in Practical: "<< pMarks<<endl;
}
};
```

```
class Result:public Theory, public Practical{
  float total;
  public:
  void display(){
   total = tMarks+ pMarks;
    Theory::display();
    Practical::display();
   cout<<"Total Marks(Theory+Practical):"<<total;
  }
};
void main(){
 Result rs;
 rs.gettMarks(40);
 rs.getpMarks(50);
 rs.display();
}
```

ii. Use *scope resolution* operator for invoking functions defined in respective classes

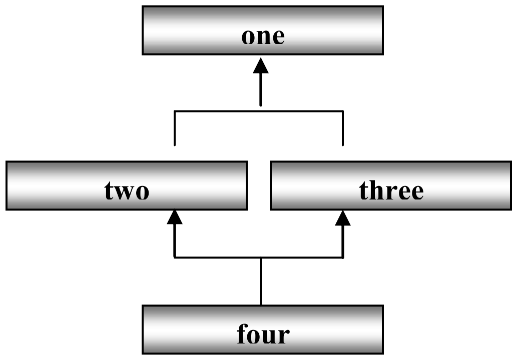
```
class one{
public:
  void display(){
   cout << "\nYou are in class ONE";
  }
};
class two{
public:
  void display(){
   cout << "\nYou are in class TWO";
  }
};
```

```
class three: public one, public two{

};

void main(){
  three obj;
  obj.one::display();   //<===
  obj.two::display();   //<===
}
```

ii. Multiple copies of the *same base class being inherited* in the *derived class*

```
class one{
  public:
    int a;
};
class two : public one{
  public:
    int b;
};
class three : public one{
  public:
    int c;
};
class four: public two, public three{
  public:
    int d;
};                 // error due to two copies of variable 'a' thru class two and three
```

```
void main(){
  four obj;
  obj.a = 999;
 //member ambigious
}
```



Here class four inherits from two base classes namely two and three which have been derived from class one so class four has two copies (duplicate sets of members) of one.

Solutions:

i. Use scope resolution operator to remove ambiguity

```
void main(){
  four obj;
  obj.two::a = 1;        // a is inherited from class two
  obj.three::a = 2;      // a is inherited from class three
  obj.d = obj.two::a +  obj.three::a;
  cout << "d is : " << obj.d;
}
```

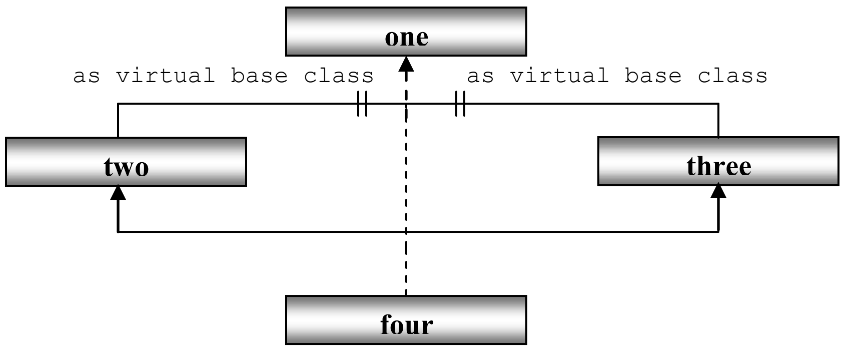
Virtual Base Class

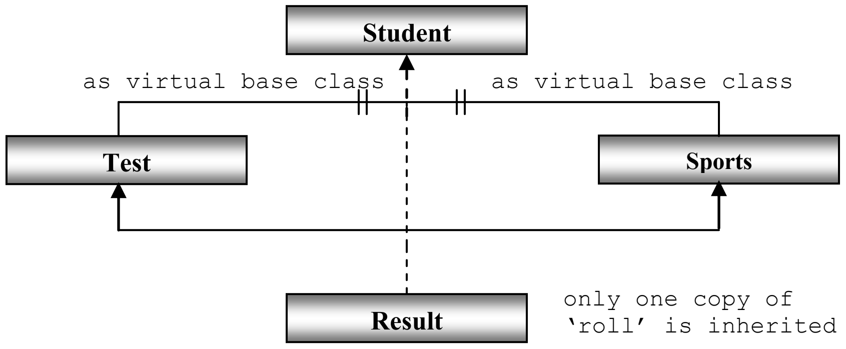
In C++, by making a class virtual **only one copy of that class is inherited** though we may have many inheritance paths between the virtual base class and a derived class
We can specify a base class inheritance by using the keyword **virtual**
The duplication of inherited members due to these multiple paths is avoided by making the common base class as virtual base class by d declaring the direct or intermediate base class
i.e.

```
class A{
 ----------------------
};
class B1:virtual public A{     //parent1
 ----------------------
};

class B2:public virtual A{     //parent2
 ----------------------
};
class C:public B1, public B2{     //child
 ----------------------
};
```

```
class one{
  public:
    int a;
};
class two : virtual public one{
  public:
    int b;
};

class three : virtual public one{
  public:
    int c;
};
```

```
class four: public two, public three{
  public:
    int d;

};
void main(){
  four obj;
  obj.a = 1;
  obj.a = 19;
  cout << “a is ” << a;
}
```





```
class Student{
  protected:
   int roll;
   public:
   void getRoll();
   void showRoll();
};
   void Student::getRoll(){
    cout<<"\n Enter student roll: ";
    cin>>roll;
   }
   void Student::showRoll(){
      cout<<"\nStudent Roll: \t"<<roll;
   }
class Test:public virtual Student{
  protected:
    float sub1, sub2;
   public:
   void getMarks();
   void showMarks();
};
   void Test::getMarks(){
    cout<<"\n Enter marks in subject1 and subject2:  ";
    cin>>sub1>>sub2;
   }
   void Test::showMarks(){
    cout<<"\nSubject1 Marks: "<<sub1<<endl<<"Subject2 Marks: "<<sub2;
   }
class Sports:public virtual Student{
  protected:
   float  score;
   public:
   void getScore();
   void showScore();
};
   void Sports::getScore(){
     cout<<"\n Enter Score: ";
     cin>>score;
   }
   void Sports::showScore(){
    cout<<"\nSports Score: \t"<<score<<endl;
   }
class Result:public Test,public Sports{
 float total;
 public:
  void displayResult();
};
 void Result::displayResult(){
    total = sub1 +  sub2 + score;
    showRoll();
    showMarks();
    showScore();
    cout<<"    Total:  \t"<<total<<endl;
    cout<<" Percentage: \t"<<(total/300) *100<<"%"<<endl;
 }
void main(){
Result rs;
 rs.getRoll();
 rs.getMarks();
 rs.getScore();
 cout<<endl<<endl<<"*****Student Result*******"<<endl;
 rs.displayResult();
 getch();
}
```

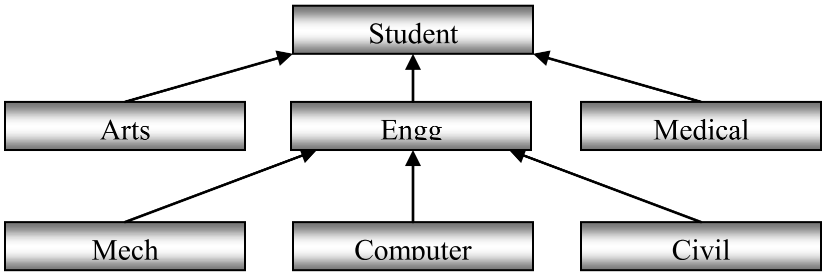
4.7.4..4 Hierarchical Inheritance

One way to use inheritance is *adding additional members* thru inheritance to extend the capabilities of a class

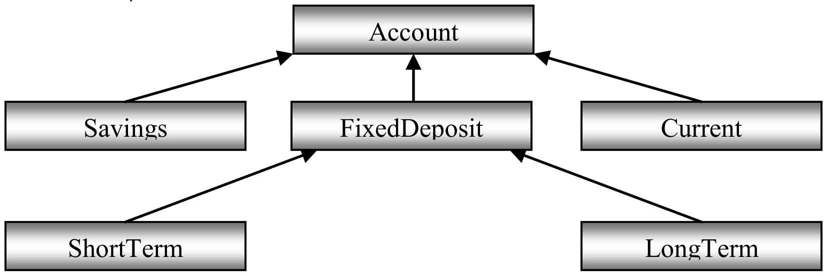
Another application of inheritance is to use it a as a *support to the hierarchical design* on a program

Cast programming problems into hierarchy by sharing features of one level by many others below that level

The *base class* will include all the features that are *common* to the subclasses and a *subclass* is constructed by inheriting the properties of *the base class*



```
class Account{
  protected:
   int id;
   char name[10];
   char address[10];
   public:
   void getInfo();
   void displayInfo();
};
   void Account::getInfo(){
    cout<<"\n Enter Name,Address and Account ID: ";
    cin>>name>>address>>id;
   }
   void Account::displayInfo(){
    cout<<"\nName: "<<name <<endl;
    cout<<"\nAddress: "<<address<<endl;
    cout<<"\nAccount ID: "<<id <<endl;
   }
class Savings:public Account{
  float interest;
  public:
   void getInfo(){
    Account::getInfo();
    cout<<"\n Enter Interest Rate: ";
    cin>>interest;
   }
   void displayInfo(){
    cout <<"\n\n *********Savings Account************"<<endl;
    Account::displayInfo();
    cout<<"\nInterest: "<<interest<< "%"<<endl;
   }
};
class Current:public Account{
  float interest;
  public:
   void getInfo(){
    Account::getInfo();
    cout<<"\n Enter Interest Rate: ";
    cin>>interest;
   }
   void displayInfo(){
    cout <<"\n\n *********Current Account************"<<endl;
    Account::displayInfo();
    cout<<"\nInterest: "<<interest<< "%"<< endl;
   }
```



```
};

class FixedDeposit:public Account{
  float amount ;
  public:
   void getInfo(){
    Account::getInfo();
    cout<<"\n Enter Amount: ";
    cin>>amount;
   }
   void displayInfo(){
    Account::displayInfo();
    cout<<"\nAmount : "<<amount<< endl;
   }
};

class ShortTerm:public FixedDeposit{
  int duration;
  float interest;
  public:
   void getInfo(){
    FixedDeposit::getInfo();
    cout<<"\n Enter Duration(months)and Interest Rate(%): ";
    cin>>duration>>interest;
   }
   void displayInfo(){
    cout <<"\n\n *********Short Term Deposit*************"<<endl;
    FixedDeposit::displayInfo();
    cout<<"\nDuration: "<<duration<<"months and  Interest: "<<interest<< "%"<<endl;
   }
};
class LongTerm:public FixedDeposit{
  int duration;
  float interest;
  public:
   void getInfo(){
    FixedDeposit::getInfo();
    cout<<"\n Enter Duration(months)and Interest Rate(%): ";
    cin>>duration>>interest;
   }
   void displayInfo(){
    cout <<"\n\n *********Long Term Deposit*************"<<endl;
    FixedDeposit::displayInfo();
    cout<<"\nDuration: "<<duration<<"months and  Interest: "<<interest<< "%"<<endl;
   }
};
main(){
 LongTerm lt;
 lt.getInfo();
 lt.displayInfo();
}
```

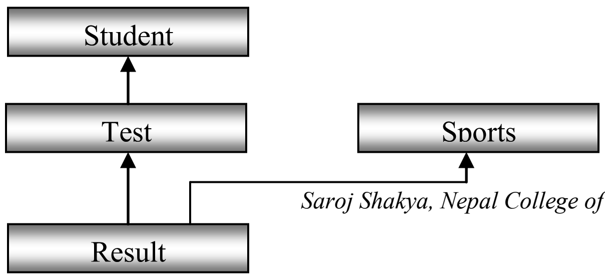
4.7.4..5 Hybrid Inheritance

Used in situations when we need to *apply two or more types of inheritance* to design a program

To design an program to process *students results,* the test results of a batch of students are stored in three different classes.

Class **Student** stores the roll number, class **Test** stores the marks obtained in two subjects, class **Result** contains the total marks obtained in the test

Assume that we have to give weightage for sports (stored in class sports) before finalizing the results



```
class student{
protected:
int roll;
public:
  void get_number(int a){
     roll = a;
  }
  void show_number(){
     cout << "roll"<< roll;
  }
};

class test : public student{
protected:
float part1,part2;
public:
  void get_marks(float x, float y){
     part1 = x; part2 = y;
  }
  void show_marks(){
     cout << "part1 "<< part1;
     cout << " part2 "<<part2;
  }
};
```

```
class sports{
protected:
  float score;
public:
 void get_score(float s){
    score = s;
 }
 void show_score(){
   cout << "sports weightage: " << score;
 }
};

class result : public test, public sports{
  float total;
  public:
 void display(){
 total = part1+part2 + score;
  show_number();
  show_marks();
  show_score();
  cout << "Total Score: "<< total;
 }
};

main(){
  result s1;
 s1.get_number(1234);
 s1.get_marks(27.5,34.5);
 s1.get_score(6.0);
 s1.display();
}
```

4.8 Constructors in Derived Classes

If the **base class does not have any constructors taking arguments** (parameterized constructors), the **derived class need not have a constructor** function

If the **base class contains a constructor with one or more arguments** then it is **mandatory for the derived class to have a constructor** and pass the arguments to the base class constructors

When both the **derived and base classes contain constructors** then the **base class constructors is execute first** and then the constructor in the derived class is executed

In case of **Multiple Inheritance**
-The **base classes are constructed in the order** in which they appear in the declaration of the derived class

In case of **Multilevel Inheritance**
-The constructors will be executed **in the order of inheritance**

Method of inheritance	Order of Execution

`Class B:public A{`
`};`
	`A(); base constructor (first)`
	`B(); derived constructor(second)`

`Class A:public B, public C{`
`}`
	`B(); base constructor (first)`
	`C(); base constructor(second)`
	`A();derived (last)`

`Class A: public B, virtual public C{`
 	
`}`
	`C(); virtual base  (first)`
	`B(); ordinary base constructor(second)`
	`A();derived (last)`

Argument passing mechanism for supplying initial values to the bases classes constructors:

The **constructor of the derived class receives the entire list of values as its arguments and passed them to the base class constructors** in the order in which they are declared in the base class

The **base class constructors are called and executed first** before executing the statements in the body of the derived constructor.

General form of defining a derived constructor:

The header line of derived constructor function contains two parts separated by a colon (:)

The first part provide the declaration of arguments that are passed to the derived constructor and the second part lists the function calls to the base constructors

```
 Derived-constructor(arglist1, arglist2, arglistD): base1(arglist1), base2(arglist2){

 }
```

Here, `base1(arglist1), base2(arglist2)` are function calls to base class constructors and `Arglist1, Arglist2` are the actual parameters that are passed to the base constructors . Here, `arglistD` provides the parameters that are necessary to initialize the members of the derived class itself.

```
class alpha{
 int a;
 public:
  alpha(int i){
   a = i;
   cout<<"alpha initializedto"<<a<< endl;
  }
  void displayAlpha(){
   cout << "alpha is "<<a<<endl;
  }
};
class beta{
 int b;
 public:
  beta(int j){
   b = j;
   cout<<"beta initialized to"<< b<<endl;
  }
  void displayBeta(){
     cout << "beta is "<<b<<endl;
    }
};
class gamma:public beta,public alpha{
  int g;
  public:
  gamma(int a, int b, int gm):  beta(b),alpha(a){
   g = gm;  // the order of arguments is not important
   cout<<"gamma initialized to"<<g<<endl;
  }
  void displayGamma(){
   cout << "gama  is "<<g<<endl;
  }
};
main(){
 gamma g(1,2,3);
 g.displayGamma();
}
```

Here, **beta is initialized first although it appears second** in the derived constructor as it has been declared first in the derived class header line

```
class gamma:public beta, public alpha{

}
```

If we change the order to

```
class gamma:public alpha, public beta{

}
```

then alpha will be initialized first

Chapter 5

5. Polymorphism

"One name multiple forms"

Polymorphic in Greek: *poly* means *many* and *morphos* means *forms*

Morphus a Greek god who could appear to sleeping individuals in any form he wished

5.1 Polymorphism in Programming Languages

In programming languages, a *polymorphic* object is any entity such as a *variable* or *function argument* that is permitted to hold values of different types during the course of execution

Polymorphic functions are those that have *polymorphic arguments*

The most common form of *polymorphism* in conventional programming languages is *overloading*, such as overloading the *+ symbol* to mean both *integer* and *real addition*

In some functional languages, *Parametric Polymorphism* is allowed which is another form in which a parameter can be characterized only partially such as list of *T* where *T* is left undefined

In OOP languages, similar features are available thru the use of generics or templates

It reflects the *principle of substitutability*, as a *polymorphic* object oriented variable is permitted to hold a value of its expected (*declared*) type or of any subtype of the expected type.

5.1.1 Compile Time Polymorphism (*Static Polymorphism*)

Refers to the *binding* of functions on the basis of their *signature* (*number , type and sequence of parameters*)

Also called *early binding* as the calls are already bound to the proper type of functions during the *compilation* of the program

Overloaded functions and *operators* support *Compile Time Polymorphism*

```
void volume (int);
void volume(int , int , int );
```

When the function `volume()` is invoked, the passed parameters determine which one is to be executed

This resolution takes place at *compile time*.

5.1.2 Runtime Polymorphism (*Dynamic Polymorphism*)

Means the change of form by an entity depending on the situation

A function exhibits dynamic polymorphism if it exists in various forms and the resolution to different functions calls are made dynamically during execution time

This feature makes the program more flexible as a function can be called depending on the context

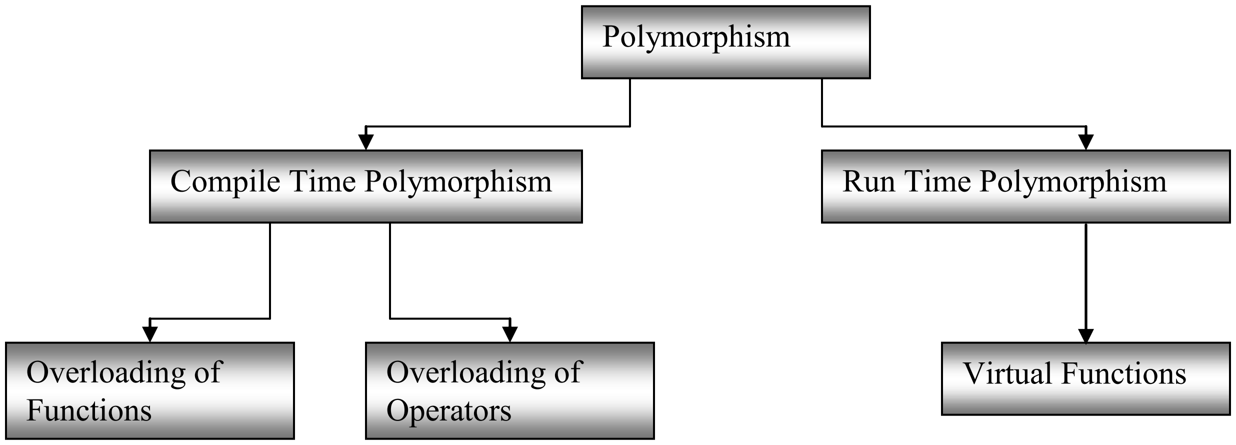
Can be achieved by using virtual functions

5.1.3 Static Polymorphism *Vs* Dynamic Binding

Dynamic Binding is more flexible and *Static Binding* is more efficient in certain cases

Statically bound functions do not require *runtime search*, while *dynamic function* calls need it

As function calls are resolved at *execution time* in case of *Dynamic Binding* the user has the flexibility to alter the call with out modifying the source code



5.2 Varieties in Polymorphism

Polymorphism is a natural result of the “*is a”* relationship and of mechanisms of *message passing , inheritance and substitutability*.

Prime concern is *code sharing* and *reuse*

Pure Polymorphism occurs when a single function can be applied to arguments of a variety of types

Here, there is one function (*code body*) and number of interpretations

Ad-hoc Polymorphism occurs when we have a number of different functions (*code bodies*) all denoted by the same name (*overloading*)

Other forms are *overriding* and *deferred methods*

5.3 Polymorphic Variables

Apart from *overloading* , polymorphism is made possible only by the existence of *polymorphic variables*

These variables can hold values of different types and represent the *principle of substitutability*

While there is an expected type for any variable the actual type can be from any value that is a subtype of the expected type.

In dynamically bound languages (*Smalltalk, Objective-C*), all variables are potentially *polymorphic* i.e. any variable can hold values of any type (*type checking is done during runtime*)

In statically bound languages (*C++, Java*), a variable can hold a value of the same type as that of the declared class of the variable or any subclass of the declared class (*type checking is done during compile time*)

5.4 Overloading

A function name is *overloaded* if there are two or more function bodies associated with it (*the same function name*)

Overloading is a necessary part of overriding

Overloading can occur with out *overriding*

In overloading, it is the function name that is *polymorphic – has many forms*

Another analogy:

There is a *single abstract function* that takes various types of arguments : *the actual code executed depends on the arguments given*

The compiler can determine the correct function at allowing it to generate single code sequence (*code optimization*)

Coerce
Parametric overloading
Another style of overloading where functions in the same context are allowed to share a name and are disambiguated by the number and type of arguments supplied called parametric overloading (similar to overloading constructors)

5.5 Overriding

Similar to *overloading* but occurs in *a hierarchical chain (different classes in different levels)*

This kind of *polymorphism* is supported by the *virtual keyword*.

A *child class* defines a method by the same name as that used for the method in the *parent class*, the method in the child class effectively *hides* or *overrides* the method in the parent class

Overriding contributes to *code sharing*

Instances of the classes that do not *override* the method can all share one copy of the original

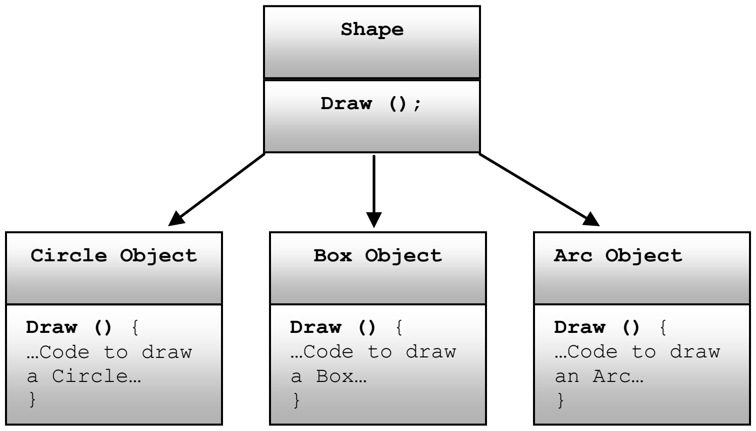
Alternative codes are written for that method only in cases where this method is not appropriate
i.e. used when a *child class* needs to *modify* the behavior of the *parent class*

5.6 Deferred Methods

A *deferred method* (sometimes called *abstract method* and in C++ called a *pure virtual method*) is a virtual function without a *body*

Very often the *virtual functions* of the *base classes* are not used

Here, the behavior described by the method in a parent class is NOT modified by the child class as the behavior of a parent class is NULL, *only just a place holder* and all useful activity is defined as part of the code provided by the child classes



5.7 Pure Polymorphism

The term *pure polymorphism* is reserved by many authors for situations where one function can be used with a variety of arguments (*generics*) and the term overloading for situations where there are multiple functions all defined with a same name

Forming *polymorphic functions* is one of the most powerful techniques in OOP that permits *code reuse*

5.8 Generics and Templates

A form of *polymorphism* provided by the facility known as a *generic* (*called a template in C++*)

Generics provide a way of *parameterizing* a class of a function by use of a type

In case of *strongly typed languages* the type can not be an unknown one e.g. the variables passed into a function or variables declared can not be an unknown type.

```
void test (unknown type x){              void test (int x){
      //  not allowed                          //type must be known
}                                        }
```

But *generics* provide this facility

With *generics* a variable is defined as a *type parameter*

This can be used within the class definition just as if it were a type although no properties of the type are known when the class description is being parsed by the compiler

The *type parameter* is matched with a specific type (*e.g. int or float etc*) and a value is declared at some later point in time

5.9 Polymorphism in C++

5.9.1 Pointers to objects (Object Pointers)

Object Pointers are useful in creating objects at *runtime*

We can also use *object pointer* to access the *public members* of an object

```
class item{
 int code; float price;
 public:

 void getdata(int a, float b){
   code = a; price = b;
 }

 void showdata(){
  cout << "Code: " << code;
  cout << "Price: " << price;
 }
};
```

```
void main(){
  item *p = new item [2];
  item *d = p;
  int x,i;
  float y;
  for(i=0;i<2;i++){
    cout << "Enter code and price for item"<< i+1;
    cin>>x>>y;
    p->getdata(x,y);
    p++;
  }
  for(i=0;i<2;i++){
    cout << "Item"<< i+1<<"\n";
    d->showdata();
    d++;
 }
}
```

5.9.2 this Pointer

"this" is a keyword to represent an object that invokes a member function, This is a pointer that *points* to the object for which this function was called. This pointer is automatically passed to a member function when it is called.

It acts as an *implicit argument* to all the member functions `e.g.`

```
class ABC{                                   This private variable 'a' can be used
  int a;                                     directly inside a member function
  -----  -----                               a = 123;    or    this->a = 123;
};
```

```
A program to compare the ages of two persons. A person class has two attributes name and
age . A constructor to initialize these attributes and a member function that accepts an
objects of the same type and compares the age of another object with its own.
```

```
class person{
   char name[20];
   float age;
 public:
   person(char *s, float a){
        strcpy(name,s);
        age=a;
   }
   int greater(person & x){
    if (x.age > this->age)
      return (x.age);
     else
    return(this->age);
    //refers to the object passing the message i.e.p1
  }
};
void main(){
  person p1("abc",104),p2("xyz",102);
   int g;
    g = p1.greater(p2);
   cout << "The bigger age is " << g;
}
```

5.9.3 Virtual Functions in C++

Virtual means existing in appearance but not in *reality*

Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message but in different forms

We use the *pointer* to *base class* to refer to all the *derived objects*

But the *base pointer* even when it is made to contain the address of a *derived class* always executes the function in the *base class*

Here, the *compiler* simply ignores the *contents of the (base) pointer* and chooses the *member function* that matches the type of the *pointer*

```
class base{
 public:
  void display(){
     cout <<" i am in base class";
  }
};
class derived: public base{
 public:
  void display(){
     cout <<" i am in derived class";
  }
};
```

```
void main(){
 base *bptr;
 derived d;
 bptr  = &d;
 bptr->display();
}
```

The output will be:

```
 i am in base class
```

When the same function name is used both in the base and derived classes the function in base class must be declared as *virtual* using the key work *virtual* preceding its normal declaration

When a function is made *virtual* , *C++* determines which function to use at *run time* based on the *type of object pointed* by the base pointer rather than the *type of pointer*

When *virtual functions* are used in a program that appears to be calling a function of one class may in reality be calling a function of a different class

The key word *virtual* tells the compiler that it should not perform *early binding* , instead it should automatically install all mechanisms necessary to perform *late binding*

This is how the objects of a *parent class* are made to access the *member functions* of the *child class objects*.

In this way *runtime polymorphism* is achieved

```
class base{
 public:
  void virtual display(){
     cout <<" i am in base class";
  }
};
class derived: public base{
 public:
  void display(){
     cout <<" i am in derived class";
  }
};
```

```
void main(){
 base *bptr;
 derived d;
 bptr  = &d;
 bptr->display();
}
```

The output will be:

```
 i am in child class
```

5.9.4 Early and Late Binding (revisited)

The *binding* refers to the *connection between* a *function call* and the *actual code executed* as a result of the *call*

If the function involved in response to each call is known at *compile time* it called *static* or *early binding* as the compiler can figure out the function to be called before the program is run

In case of *Dynamic Binding* , the actual function called at *runtime* depends on the *contents of the pointer*

It is also known as *late binding* as the *connection between* the *function call* and the *actual code executed* by the *call* is determined *late* during the *execution* of the program and NOT when the program is *compiled*

5.9.5 Rules for Virtual Functions

The following rules should be taken into account while using virtual functions for implementing late binding:

i. The *virtual functions* must be members of some class

ii. These cannot be *static member functions*

iii. These can be accessed by using *pointers to objects*

iv. These can be *friend* of some other classes

v. These must be defined in *base class* , even if you do not use it

vi. These must have same *prototypes* (*same function name with same number/type of arguments*) in both the *base* and all the *derived classes*, if not these will be considered as *overloaded functions* and *virtual function* mechanism is ignored

vii. We cannot have *virtual constructors* but we can have *virtual destructors*

viii. The *base class pointer* can point to any type of the *derived objects* but the *derived class pointer* cannot access the *base class objects* i.e. we cannot use a pointer to a derived class to access an object of the base type

ix. When a *base pointer* points to a *derived class* , incrementing or decrementing it will not make it to point to the next object of the derived class. It takes place *relative* to its base type
So we should not use this method to move the pointer to the next object

x. In case we do not redefine a *virtual function* in the derived class , the base class function will be called

5.9. 6 Pure Virtual Function

The *virtual functions* are declared *inside the base class* and *redefined* in the *derived class*

The function inside the *base class* is seldom used for performing any tasks

It serves only as a *placeholder*

Such functions are called "*do nothing*" functions or "*pure virtual functions"*

```
virtual void display() =0;
```

A *pure virtual function* is a function declared in a base class that has *no definition* (*implementation/body*) relative to the base class

The *child classes* are allowed to *inherit* them

In this situation, the compiler requires each derived class to either define the function or re-declare it as a *pure virtual function*

5.9.7 Abstract Class

A class having *at least one pure virtual function* is called an *abstract class*

No objects can be created for the *abstract class*

But *pointers to abstract class* can be created for selecting the proper *virtual function*

The main objective of an *abstract base class* is

To *derive specific classes* of the same kind and traits

To *create a base pointer* required for achieving *runtime polymorphism*

```
class base{
 public:
  void virtual message()= 0;
};
class derived1: public base{
 public:
  void message(){
     cout <<" i am in derived class 1";
  }
};
class derived2: public base{
 public:
  void message(){
     cout <<" i am in derived class 2";
  }
};
```

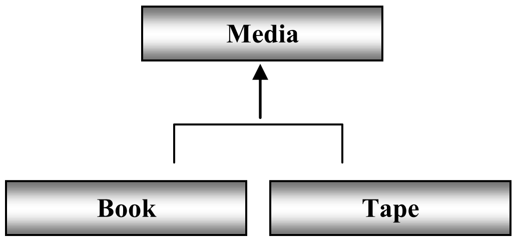
```
void main(){
 base *bptr[2];
 derived1 obj1; derived2 obj2;
 bptr[0]=&obj1;
 bptr[1]=&obj2;
 bptr[0]->message();
 bptr[1]->message();
}
```

The output will be:

```
 i  am in derived class 1
 i  am in derived class 2
```

A bookshop sells both books and video tapes. Create an abstract class known as media that stores the title and price of a publication. Create two child classes one for storing the number of pages in a book and another for storing the playing time of a tape. A function display is used in all the classes to display the class contents. Create necessary constructors in the child classes to store the information. In the main display the information regarding the book and tape using the base pointer (an object pointer of the class media).

```
class media{
  protected:
   char   title[20]; float   price;
  public:
  media(char *tt, float pp){
    strcpy(title,tt);
    price = pp;
  }
 virtual void display() = 0;
};
class book : public media{
  int  pages;
  public:
   book(char *t, float p, int pa):media(t,p){
     pages = pa;
   }
   void display( ){
     cout <<"Book Name: "<<title<<endl;
     cout <<"Book Price: "<<price<<endl;
     cout <<"Book Pages: "<<pages<<endl;
   }
};
class tape : public media{
  int  time;
  public:
   tape(char *t, float p, int tm):media(t,p){
     time = tm;
   }
   void display( ){
     cout <<"Tape Name: "<<title<<endl;
     cout <<"Tape Price: "<<price<<endl;
     cout <<"Tape Time: "<<time<<endl;
   }
};
```



```
void main(){
 media *m[2];
  book b("OOP",120.25,999);
  tape t("OODM",190.25,60);
  m[0] = &b;
  m[0]->display();
 getch();
  m[1] = &t;
  m[1]->display();
 getch();
}
```

Generic Programming
Templates

A new concept that enables us to define *generic classes* and *functions* and thus provides support for *generic programming*

Generic programming is an approach where *generic types* are used as *parameters* in algorithms so that they work for a variety of suitable data types and data structures

A *template* can be used to create a family of classes or functions

e.g. a *class template* for an array class enables us to create array of various data types such as *int* or a *float* array

e.g. a *template* for a function *mul()* that multiplies *int*, *float*, *double* types

A *template* is a kind of a *macro*

When an object of a specific type is defined for actual use the template definition for that class is substituted with the required data type

A *template* is defined with a *parameter* that would be replaced by a specified data type at the time of actual use of the class or function

So a template is also called a *parameterized class* or *functions*

The general format of a *class template*:

```
template <class T>
class classname{
   //class member specification
   //with anonymous type T
   //wherever appropriate
};
```

e.g

A regular class to store an array of *int* numbers and perform *scalar product* of two *int* vectors:

```
class vector{
private:
 int *v;
 int size;

public:

 vector(int m){
  v = new int [size = m];
  for(int i=0;i<size; i++)
    v[i] = 0;
  }

 vector(int *a){
  for(int i=0;i<size; i++)
      v[i] = a[i];
  }
```

```
int operator* (vector &y){
 int sum = 0;
  for(int i=0;i<size; i++)
      sum +=  this->v[i] * y.v[i];
   return sum;
  }
};

int main(){
  int x[3] = {1,2,3};
  int y[3] = {4,5,6};
  vector v1(3);
  vector v2(3);
  v1 = x;
  v2 = y;
  int R = v1 * v2;
  cout << "R= " << R;
}
```

A vector class to store an array of *int* numbers and perform *scalar* product of two *int* vectors:

```
template<class T>
class vector{
private:
 T *v;
 int size;
public:
 vector(int m){
  v = new T [size = m];
  for(int i=0;i<size; i++)
    v[i] = 0;
  }
 vector(T *a){
  for(int i=0;i<size; i++)
      v[i] = a[i];
  }
```

```
T operator* (vector &y){
 T sum = 0;
  for(int i=0;i<size; i++)
      sum +=  this->v[i] * y.v[i];
   return sum;
  }
};
int main(){
  int x[3] = {1,2,3};
  int y[3] = {4,5,6};
  vector <int> v1(3);
  vector <int> v2(3);
  v1 = x;
  v2 = y;
  int R = v1 * v2;
  cout << "R= " << R;
}
```

The prefix *template <class T>* tells the *compiler* hat we are going to declare a *template* and use *T* as the *type name* in the declaration

T may be substituted by any data type including the *user defined types*

The syntax for defining an object of a *template class* (*instantiation*):

```
classname <type> objectname (arglist);
vector <int>     v1(10);
vector <float>   v1(10);
Vector <complex> v3(5);
```

5.9. 8.1 Class templates with multiple parameters

More than one *generic data types* (*declared as a comma separated list within the template specification*) in a class template can be used.

The general format:

```
template <class T1, class T2, ……>
class classname{
   ……………………
   //(body of class)
   ……………………
};
```

e.g.

```
template<class T1,class T2>
class test{
private:
 T1 a;      T2 b;
public:
 test(T1 x,T2 y){
   a = x;      b = y;
 }
 void show(){
  cout << a << " and " << b;
 }
};
```

```
int main(){
  test <float, int> test1(1.23,123);
  test <int, char> test2(100,'w');
  test1.show();
  test2.show();
}
```

Template classes and *functions* eliminate *code duplication* for different types and thus make the program development easier and more manageable

5.9. 9 Function Templates

Function templates can be used to create a family of functions with different argument types
The syntax

```
template <class T>
return_type function_name(arguments_of_type T){
   ........................
   //body of function with type T
   //wherever appropriate
   ........................
};
```

Here we are defining *functions* instead of *classes*

We are using the *template parameter T* when necessary inside the *function body* and in its *argument list*

A *swap()* function template that will swap two values of a given type of data:

```
template <class T1>
void  swap(T &x, T &y){
 T temp;
   x = temp;
   x = y;
   y = temp;
};
```

This declares a set of *overloaded functions* one for each type of data

The function *swap( )* can be invoked in similar manner:

```
A program to swap two set of values(integer and float) using function templates.
```

```
template <class T>
void  swap(T &x, T &y){
 T temp = x ;
   x = y;
   y = temp;
};
void fun (int m, int n, float a, float b){
  cout << "m,n before swap" << m<<" " << n;
  swap (m,n);
  cout << "m,n after swap" << m<<" " << n;
```

```
cout << "a,b before swap" << a <<" " << b;
swap (a,b);
cout << "a,b after swap" << a <<" " << b;
}

void main(){
 fun(100,200,11.22,13,99);
}
```

5.9. 9.1 Overloading of template functions

A template function may be *overloaded* either by template function or ordinary function of its name

Overloading resolution is accomplished as follows

i. Call an *ordinary function* that has an exact match
ii. Call a *template function* that could be created with an exact match
iii. Try *normal overloading resolution* to ordinary functions and call the one that matches

No automatic conversions are applied to arguments on the template functions

A program to overload a template function with an *explicit function*

```
template <class T>
void display(T x){
 cout << "Template Display " << x ;
};
void display(int x){
 cout << "Explicit Display " << x ;
};
```

```
main(){
 display(100);
 display(1.12);
 display('a');
}
```

The call *display(100)* invokes *ordinary* version of *display()* and not the *template version*

A program for finding the highest value of the two (integers, float etc) using function templates that return a value

```
template <class T>
T  max(T x, T y){
  return x > y ? x : y;
};
```

```
main(){
 float m;
 m = max(4.5,6.5);
 cout << "The maximum value is: " << m;
}
```

5.9. 9.2 Function Templates with multiple arguments

More than one *generic data type* in the template statement using a *comma separated list* can be used for constructing function templates with *more than one parameter*

```
template <class T1, class T2>
void display(T1 x, T2 y){
 cout << x << y ;
};
```

```
main(){
 display(199,”ere”);
 display(199,1.2);
}
```

5.9. 9.3 Member function Templates

In earlier cases, all member functions were defined as *inline*

When they are defined outside the class these functions must be defied by the *function templates*

```
The Syntax:

template <class T>
return_type class_name<T>:: function_name(argument_list){
   ........................
   //body of function
   ........................
};
```

```
The vector class with member functions defined outside the class:
template<class T>
class vector{
private:
 T *v;
 int size;
public:
 vector(int m);
 vector(T *a);
 T operator* (vector &y);
};
//Member function templates
template<class T>
vector<T>::vector(int m){
  v = new T [size = m];
  for(int i=0;i<size; i++){
    v[i] = 0;
  }
 template<class T>
 vector<T>::vector(T *a){
```

```
  for(int i=0;i<size; i++)
      v[i] = a[i];
  }
template<class T>
T vector<T>:: operator* (vector &y){
 T sum = 0;
  for(int i=0;i<size; i++)
      sum +=  this->v[i] * y.v[i];
   return sum;
  }
int main(){
  int x[3] = {1,2,3};
  int y[3] = {4,5,6};
  vector <int> v1;
  vector <int> v2;
  v1 = x;
  v2 = y;
  int R = v1 * v2;
  cout << “R= ” << R;
}
```

5.9. 9.4 Non type Template Arguments

In addition to the *type argument ‘T’* we can also use other arguments such as *string, function names, constant expressions and built in types*

```
template <class T, int size>
class <T>{
  T a[size];
  ........................
};
```

This template supplies the size of the array as an argument

So we can make

```
array <int, 10>   a1; // array of 10 integers
array <float, 10> a2; // array of 10 floats
array <char, 20>  a3; // string of size 20
```

The size is given as an argument to the template class

6. Exception Handling

Logic errors (Semantic): Due to poor understanding of the *problem* and *solution* procedure

Syntactic errors (Syntax): Due to poor understanding of the *language* itself

We can detect these errors by using *exhaustive debugging* and *testing* procedures

Peculiar problems *other than logic or syntax errors* is known as *exceptions*

They are *runtime anomalies* or *unusual conditions* that a program may encounter *while executing*
e.g. *division by zero*, *access to an array outside of its bounds* , *running out of memory or disk space*

These exceptional conditions must be identified and dealt with effectively

Exception handling provides type-safe integrated approach for coping with unusual predictable problems that arise while executing a program

Basics

There are two kinds of exceptions:

Synchronous

- Errors such as "*out of range index*" and "*over flow*"

Asynchronous

- Errors that are caused by events beyond the control of the program (*e.g. keyboard interrupts*)

The purpose of exception handling mechanism is to *provide means to detect* and *report an "exceptional circumstance"* so that appropriate action can be taken

The mechanism suggests a separate error handling code that performs the following tasks:

i. Find the problem (*Hit the exception*)
ii. Inform that an error has occurred (*Throw the exception*)
iii. Receive the error information (*Catch the exception*)
iv. Take the corrective actions (*Handle the exception*)

Error handling code basically consists of two segments

- One to detect errors and to throw exceptions
- The other to catch the exception and to take appropriate actions

6.1 Exception Handling Mechanism

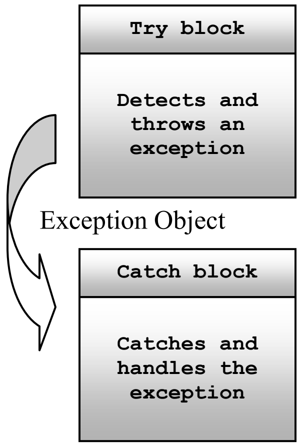
Comprises of basically three keywords *try, throw* and *catch*

Try is used to preface a block of statements surrounded by braces, which may generate exceptions

This block of statements is known as *try block*

When an exception is detected, it is thrown using a *throw statement* in the *try block*

The *catch block* catches and handles the exception thrown by the *throw statement* in the *try block*


Try block
Detects and throws an exception
Exception Object
Catch block
Catches and handles the exception

```
General format:
……………
……………
try {
    …………
 throw exception; // block of statements
                  //which    detects    and
                  //throws an exception
    …………..
}
catch (type arg){  // catches exception
 …………..           //block of statements
                   //that handles
}
……………
```

The program control leaves the try block and enters the catch statement of the catch block when the try block throws an exception

Exceptions are objects used to *transmit information about a problem*

The catch block is executed for handling the exception if the type of object thrown matches the *arg* type in the catch statement

If they do not match then the program is aborted with the help of the abort function invoked by *default*

If no exception is detected and thrown the control goes to the statement immediately after the *catch block*

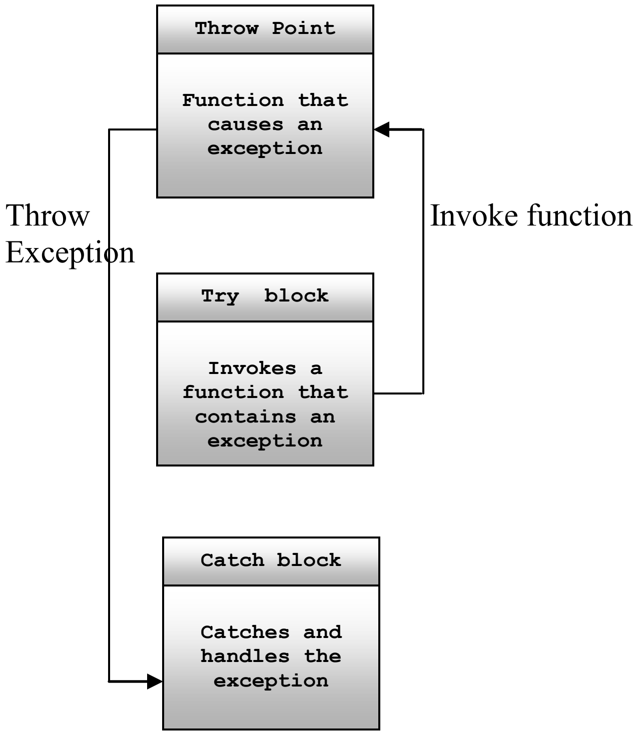
```
void main(){
int a,b;
cout <<"Enter values of a and b\n";
cin >> a;   cin >> b;
int x = a - b;
try{
  if(x != 0){
    cout << "Result (a/x) = " << a/x << "\n";
     // no error
 }
 else{      //   there  is  an  exception  /
division by zero
   throw(x);//  throws an int object
 }
}
catch(int i){
 cout<< "Exception caught: x = " << x<< "\n";
}
cout << "END";
}
```

Here, the program detects and catches a *division by zero problem*

Most often exceptions are thrown by function that are invoked from within the try blocks

The point at which the throw is executed is called the *throw point*

Once the exception is thrown to the catch block, control cannot return to the throw point


Throw Point
Function that causes an exception
Throw Exception
Invoke function
Try block
Invokes a function that contains an exception
Catch block
Catches and handles the exception

Function invoked by try block throwing exception

```
General format:

Type function (arg list){
               // function with exception
……………
 throw(object);//throws exception
……………
}

try {
    …………
            // invoke function here
    …………
}

catch (type arg){  // catches exception
 …………              //block of statements
                   //that handles
}
…………
```

An example showing an exception thrown inside a function and caught in main.

```
void divide (int x, int y , int z){
  cout << "we are inside the function\n";
  if( ( x - y ) != 0 ){
     int R = z / (x - y);
     cout << "Result  = " << R << "\n";
  }
 else{   //there is a problem
     throw(x - y);   //throw point
 }
}
```

```
void main(){
try{
 cout << "we are inside the try block\n";
  divide (10,20,30);
  //divide (10,10,30);
}
 catch(int i){ //  catches the  exception
 cout << "Exception Caught div by " << i;
}
getch();
}
```

6.2 Throwing Mechanism

Forms of throw statement

```
throw(exception);       throw exception;        throw;
```

The operand object exception may be of any type including constants

When an exception is thrown it will be caught by the catch statement associated with the try block

Control exists the current try block and is transferred to the catch block after that try block

6.3 Catching Mechanism

Code for handling exceptions is included in catch blocks

```
catch(type arg){
    // statements for managing exceptions
}
```

type is the type of exception that it handles and *arg* is an *optional parameter name*
catch statement catches an exception whose type matches with the type of *catch argument*

When it is caught the code in the *catch block* is executed

6.4 Multiple Catch Statements

It is possible for a program segment to have *more than one condition* to throw an exception

We can associate more than one catch statement with a *try statement*

An example of function invoked by a try block throwing an exception

```
General format:
try {
    …………
            // try block
    …………
}
catch (type1 arg){
 …………               //catch block1
}
catch (type2 arg){
 …………               //catch block2
}
catch (typeN arg){
 …………               //catch blockN
}
```

On encountering an exception:

- The exception handlers are searched in order for an appropriate match
- The first handler that matches the exception type is executed
- Upon executing the handler, all other handlers are bypassed and control goes to the first statement after the last catch block for that try
- If no match is found the program is terminated

```
void test (int x){
try{
    if(x == 1) throw x;
  else
    if(x == 0) throw ‘x’;
  else
   if(x == -1)throw 1.0;
   cout << “End of try block \n”;
}
 catch(int c){    //  catch 1
   cout << "caught  a character ";
}
catch(int m){     //  catch 2
   cout << "caught  an integer ";
}
catch(double d){  //  catch 3
   cout << "caught  a double ";
}
  cout << “End of try-catch system \n”;
}
void main(){
   cout << “testing multiple catches \n”
   cout << “x == 1 \n”;
   test(1);
   cout << “x == 0 \n”;
   test(0);
 getch();
}
```

6.5 Catching all types of Exceptions

In situations when we may not be able to anticipate all possible types of exceptions and design independent catch handlers to catch them

Only one catch statement to catch all exceptions instead of a certain type alone

```
Syntax:
catch(…){   //three dots inside a pair of braces
 ……………      // Statement for processing all exceptions
}
```

A program to with one catch statement to catch all types of exceptions

```
void test (int x){
try{
    if(x == 1) throw x;
    if(x == 0) throw 'x';
    if(x == -1)throw 1.0;
  }
catch(...){ //  catch all
 cout << "caught  an exception ";
}
}
void main(){
   cout << "testing Generic catch \n" ;
   test(1);   test(0);
}
```

Stack

```
#define max 10
class stack{
 int stk[max];
 int top;
 public:
  stack(){
  top = -1;
  }

  void push(int data){
   if (top == (max -1))
    cout << "stack is full\n";
    else{
     top++;
     stk[top]= data;
    }
   }
   void pop(){
   if (top == -1)
    cout << "stack is empty\n";
    else{
     int data = stk[top];
     top--;
    }
   }
   void show(){
    for(int i=top;i>=0;i--)
    cout << "\nstack["<< i <<"]" << stk[i];
   }
};

void main(){
 stack s;
 s.push(11); s.push(112);
 s.push(114); s.push(15);
 s.show();
 getch();
cout << "\n  poped top\n";
 s.pop();
 s.show();
 getch();
}
```

Stack using Class Template.

```
#define max 10
template <class T>
class stack{
 T stk[max];
 int top;
 public:
  stack(){
  top = -1;
  }

  void push(T data){
   if (top == (max -1))
    cout << "stack is full\n";
    else{
     top++;
     stk[top]= data;
    }
   }
   void pop(){
   if (top == -1)
    cout << "stack is empty\n";
    else{
     top--;
    }
   }
   void show(){
    for(int i=top;i>=0;i--)
    cout << "\nstack["<< i <<"]" << stk[i];
   }
};

void main(){
 stack <char> s;
 s.push('a');
 s.push('b');
 s.push('d');
 s.push('e');
 s.push('f');
 s.show();
 getch();
 cout << "\n poped top\n";
 s.pop();
 s.show();
 getch();
}
```