CS304-Handouts

© Virtual University of Pakistan

LECTURE NO.01 8

01.1. INTRODUCTION 8
01.2. WHAT IS A MODEL? 10
01.3. OO MODELS: 11
01.4. OBJECT-ORIENTATION - ADVANTAGES 12
01.5. WHAT IS AN OBJECT? 12
01.6. TANGIBLE AND INTANGIBLE OBJECTS 13
01.7. SUMMARY: 14

LECTURE NO.02 15

02.1. INFORMATION HIDING: 15
02.2. ENCAPSULATION 16
02.3. INTERFACE 17
02.4. IMPLEMENTATION 18
02.5. SEPARATION OF INTERFACE & IMPLEMENTATION 19
02.6. MESSAGES 19
02.7. SUMMARY 19

LECTURE NO.03 21

03.1. ABSTRACTION 21
03.2. CLASSES 23
03.3. INHERITANCE 25

LECTURE NO.04 29

04.1. CONCEPTS RELATED WITH INHERITANCE 29
04.2. GENERALIZATION 29
04.3. SUB-TYPING (EXTENSION) 31
04.4. SPECIALIZATION (RESTRICTION) 32
04.5. OVERRIDING 34
04.6. ABSTRACT CLASSES 36
04.7. CONCRETE CLASSES 38

LECTURE NO.05 40

05.3. SIMPLE ASSOCIATION 50
05.4. COMPOSITION 52
05.5. AGGREGATION 54

LECTURE NO.06 55

06.1. CLASS COMPATIBILITY 55
06.2. POLYMORPHISM 56
06.3. POLYMORPHISM IN OO MODEL 56
06.4. POLYMORPHISM – ADVANTAGES 57
06.5. OBJECT-ORIENTED MODELING AN EXAMPLE 57

LECTURE NO.07 65

07.1. CLASS 65
07.2. TYPE IN C++ 65
07.3. ABSTRACTION 66
07.4. DEFINING A NEW USER DEFINED TYPE 66
07.5. OBJECT AND CLASS: 68
07.6. ACCESSING MEMBERS 68
07.7. ACCESS SPECIFIERS 69

LECTURE NO.08 72

 © Virtual University of Pakistan

08.1. MEMBER FUNCTIONS 72
08.2. DEFINING MEMBER FUNCTIONS 72
08.3. INLINE FUNCTIONS 73
08.4. CONSTRUCTOR 75
08.5. CONSTRUCTOR PROPERTIES 75
08.6. DEFAULT CONSTRUCTOR 76
08.7. CONSTRUCTOR OVERLOADING 77
08.8. CONSTRUCTOR OVERLOADING 78
08.9. COPY CONSTRUCTOR 79
08.10. SHALLOW COPY 81
08.11. DEEP COPY 82

LECTURE NO.09 84

09.1. SHALLOW COPY 85
09.2. DEEP COPY 90
09.3. IMPORTANT POINTS ABOUT COPY CONSTRUCTOR: 94
09.4. DESTRUCTOR 94
09.5. ACCESSOR FUNCTIONS 95
09.6. THIS POINTER 96

LECTURE NO.10 99

10.1. USES OF THIS POINTER 99
10.2. SEPARATION OF INTERFACE AND IMPLEMENTATION 99
10.3. COMPLEX NUMBER 100
10.4. CONST MEMBER FUNCTIONS 102
10.5. THIS POINTER AND CONST MEMBER FUNCTION 104

LECTURE NO.11 105

11.1. USAGE EXAMPLE OF CONSTANT MEMBER FUNCTIONS 105
11.2. DIFFERENCE BETWEEN INITIALIZATION AND ASSIGNMENT: 106
11.3. MEMBER INITIALIZER LIST 106
11.4. CONST OBJECTS 107
11.5. STATIC VARIABLES 109

LECTURE NO.12 112

12.1. ACCESSING STATIC DATA MEMBER 113
12.2. LIFE OF STATIC DATA MEMBER 114
12.3. STATIC MEMBER FUNCTION 115
12.4. THIS POINTER AND STATIC MEMBER FUNCTIONS 116
12.5. GLOBAL VARIABLE VS. STATIC MEMBERS 116
12.6. ARRAY OF OBJECTS 116

LECTURE NO.13 118

13.1. POINTER TO OBJECTS 118
13.2. BREAKUP OF NEW OPERATION 119
13.3. CASE STUDY 119
13.4. COMPLETE CODE OF DATE CLASS 121

LECTURE NO.14 124

14.1. COMPOSITION 124

LECTURE NO.15 132

15.1. AGGREGATION 135
15.2. FRIEND FUNCTIONS 138

LECTURE NO.16 142

16.1. OPERATOR OVERLOADING 142
LECTURE NO.17 149
17.1. OVERLOADING ASSIGNMENT OPERATOR 151
LECTURE NO.18 155
18.1. SELF ASSIGNMENT PROBLEM: 155
18.2. OTHER BINARY OPERATORS 156
18.3. FRIEND FUNCTIONS AND OPERATOR OVERLOADING 157
LECTURE NO.19 158
19.1. STREAM INSERTION OPERATOR 158
19.2. STREAM EXTRACTION OPERATOR 158
19.3. OVERLOADING STREAM INSERTION OPERATOR 159
19.4. OVERLOADING STREAM EXTRACTION OPERATOR: 160
19.5. OTHER BINARY OPERATORS: 161
LECTURE NO.20 163
20.1. SUBSCRIPT [] OPERATOR 164
20.2. OVERLOADING SUBSCRIPT [] OPERATOR 164
20.3. OVERLOADING FUNCTION () OPERATOR 165
20.4. FUNCTION OPERATOR PERFORMING SUB STRING OPERATION, ... 165
20.5. UNARY OPERATORS 166
LECTURE NO.21 168
21.1. BEHAVIOR OF ++ AND -- FOR PRE-DEFINED TYPES: 168
21.2. POST-INCREMENT OPERATOR: 169
21.3. TYPE CONVERSION 170
21.4. USER DEFINED TYPES: 173
21.5. DRAWBACKS OF TYPE CONVERSION OPERATOR: 174
LECTURE NO.22 175
22.1. PRACTICAL IMPLEMENTATION OF INHERITANCE IN C++ 175
22.2. INHERITANCE IN CLASSES 175
22.3. UML NOTATION 175
22.4. INHERITANCE IN C++ 175
22.5. "IS A" RELATIONSHIP 176
LECTURE NO.23 183
23.1. ACCESSING BASE CLASS MEMBER FUNCTIONS IN DERIVED CLASS: 183
23.2. "*PROTECTED*" ACCESS SPECIFIER: 185
23.3. "IS A" RELATIONSHIP 186
23.4. STATIC TYPE 189
LECTURE NO.24 191
24.1. MODIFIED DEFAULT CONSTRUCTOR 199
LECTURE NO.25 207
25.1. OVERLOADING VS. OVERRIDING 207
25.2. HIERARCHY OF INHERITANCE 211
LECTURE NO.26 213
26.1. BASE INITIALIZATION 213
26.2. TYPES OF INHERITANCE 214
26.3. PRIVATE INHERITANCE 216

 © Virtual University of Pakistan

LECTURE NO.27 218

27.1. SPECIALIZATION (RESTRICTION) 218
27.2. PROTECTED INHERITANCE 222
27.3. PROPERTIES OF PROTECTED INHERITANCE 223

LECTURE NO.28 225

28.1. VIRTUAL FUNCTIONS 225
28.2. VIRTUAL FUNCTIONS: 230
28.3. SHAPE HIERARCHY 230
28.4. STATIC VS DYNAMIC BINDING 231

LECTURE NO.29 233

29.1. ABSTRACT CLASSES 233
29.2. CONCRETE CLASSES 234
29.3. ABSTRACT CLASSES IN C++ 234
29.4. PURE VIRTUAL FUNCTIONS 234
29.5. SHAPE HIERARCHY 235
29.6. VIRTUAL DESTRUCTORS 236
29.7. VIRTUAL FUNCTIONS – USAGE 238
29.8. V TABLE 240
29.9. DYNAMIC DISPATCH (DYNAMIC BINDING) 242

LECTURE NO.30 243

30.1. POLYMORPHISM – CASE STUDY: A SIMPLE PAYROLL APPLICATION 243
30.2. SHAPE HIERARCHY REVISITED: 246

LECTURE NO.31 250

31.1. MULTIPLE INHERITANCE 250
31.2. PROBLEMS IN MULTIPLE INHERITANCE 251
31.3. VIRTUAL INHERITANCE 256

LECTURE NO.32 258

32.1. GENERIC PROGRAMMING 259
32.2. TEMPLATES 259
32.3. FUNCTION TEMPLATES 260

LECTURE NO.33 264

33.1. MULTIPLE TYPE ARGUMENTS 264
33.2. USER-DEFINED TYPES 264
33.3. OVERLOADING VS. TEMPLATES 265
33.4. TEMPLATE ARGUMENTS AS POLICY: 266
33.5. FIRST SOLUTION: 267
33.6. SECOND SOLUTION: 267
33.7. THIRD SOLUTION 268
33.8. DEFAULT POLICY 269

LECTURE NO.34 270

34.1. GENERIC ALGORITHMS 270
34.2. CLASS TEMPLATES 272
34.3. EXAMPLE – CLASS TEMPLATE 273

LECTURE NO.35 276

35.1. MEMBER TEMPLATES: 276
35.2. CLASS TEMPLATE SPECIALIZATION 278

LECTURE NO.36 282

36.1. MEMBER TEMPLATES REVISITED 282
36.2. PARTIAL SPECIALIZATION: 283
36.3. FUNCTION TEMPLATES 284
36.4. COMPLETE SPECIALIZATION 285
36.5. USING DIFFERENT SPECIALIZATIONS 286
36.6. NON-TYPE PARAMETERS 286
36.7. EXAMPLE – TEMPLATE CLASS ARRAY 286
36.8. DEFAULT NON-TYPE PARAMETERS 288
36.9. DEFAULT TYPE PARAMETERS 288

LECTURE NO.37 288

37.1. RESOLUTION ORDER 289
37.2. FUNCTION TEMPLATE OVERLOADING 290
37.3. RESOLUTION ORDER 291
37.4. TEMPLATES AND INHERITANCE 291
37.5. DERIVATIONS IN CASE OF A GENERAL TEMPLATE CLASS 291

LECTURE NO.38 296

38.1. TEMPLATES AND FRIENDS 296
38.2. TEMPLATES AND FRIENDS – RULE 1 296
38.3. TEMPLATES AND FRIENDS – RULE 2 297
38.4. TEMPLATES AND FRIENDS – RULE 3 299
38.5. TEMPLATES AND FRIENDS – RULE 4 301

LECTURE NO.39 303

39.1. TEMPLATES & STATIC MEMBERS 303
39.2. TEMPLATES – CONCLUSION 304
39.3. GENERIC ALGORITHMS REVISITED 305
39.4. GENERIC ALGORITHMS REVISITED 306
39.5. GENERIC ALGORITHM 308
39.6. PROBLEMS 308

LECTURE NO.40 309

40.1. CURSORS 309
40.2. ITERATORS 312

LECTURE NO.41 316

41.1. STANDARD TEMPLATE LIBRARY: 316
41.2. STL CONTAINERS 316
41.3. COMMON FUNCTIONS FOR ALL CONTAINERS 322
41.4. FUNCTIONS FOR FIRST-CLASS CONTAINERS 322
41.5. CONTAINER REQUIREMENTS 323

LECTURE NO.42 325

42.1. ITERATORS 325
42.2. ITERATOR CATEGORIES 325
42.3. ITERATOR SUMMARY: 325
42.4. CONTAINER AND ITERATOR TYPES: 326
42.5. SEQUENCE CONTAINERS 326
42.6. ASSOCIATIVE CONTAINERS 327
42.7. CONTAINER ADAPTERS 327
42.8. ITERATOR OPERATIONS 327
42.9. ALGORITHMS 331

LECTURE NO.43 334

43.1. EXAMPLE – ABNORMAL TERMINATION 334

 © Virtual University of Pakistan

43.2. GRACEFUL TERMINATION 335
43.3. ERROR HANDLING 335
43.4. EXCEPTION HANDLING 338
43.5. EXCEPTION HANDLING PROCESS 338
LECTURE NO.44 ..... **342**
44.1. STACK UNWINDING 342
LECTURE NO.45 ..... **349**
45.1. RESOURCE MANAGEMENT 349

Lecture No.01

01.1. Introduction

Course Objective:
Objective of this course is to make students familiar with the concepts of object oriented programming. These concepts will be reinforced by their implementation in C++.

Course Contents:
The main topics that we will study in the 45 lectures of this course are given below,

- Object Orientation
- Objects and Classes
- Overloading
- Inheritance
- Polymorphism
- Generic Programming
- Exception Handling
- Introduction to Design Patterns

Recommended Text Book:

C++ How to Program (Deitel & Deitel **)**

Reference Books:

1. **Object-Oriented Software Engineering**
   By Jacobson, Christerson, Jonsson, Overgaard
   (For object oriented programming introductory concepts)
2. **The C++ Programming Language**
   By Bjarne Stroustrup
   (For better c++ understanding)

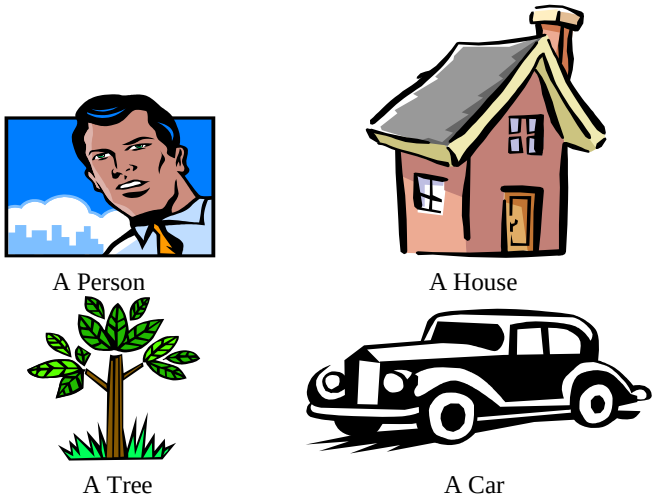
© Virtual University of Pakistan

Object-Orientation (OO)

What is Object-Orientation?

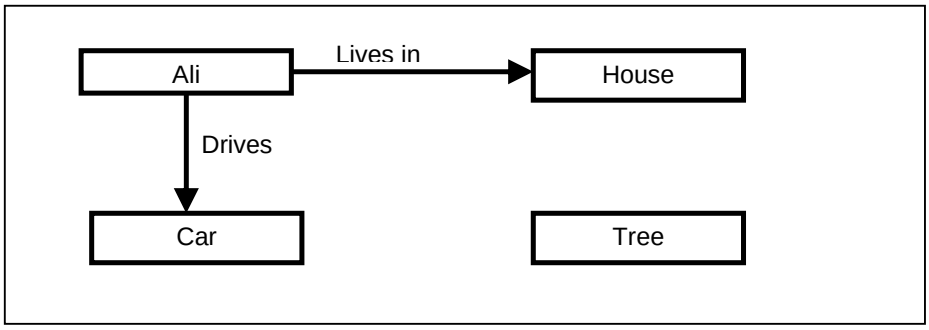
It is a technique in which we visualize our programming problems in the form of objects and their interactions as happens in real life.

Examples:
We have different objects around us in our real life that interact with each other to perform different operations for example,



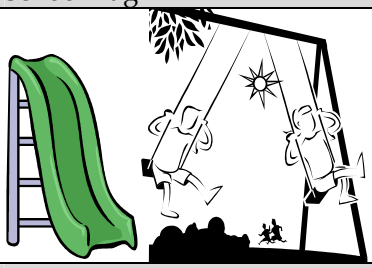
Different Objects

These objects interact with each other to perform different operations,



Take another example of a **School**; the objects in a school are **student**, **teacher**, **books, pen ,school bag**, **classroom, parents, playground and so on… ,**

Objects in a School

		
Teacher	Student	School Bag
		
Book	Pen	Playground
		
Parents	Classroom	Library

Suppose we want to develop a fee collection system for a school for this we will need to find out related objects and their interactions as happens in real life.
In this way we can say that **object orientation** makes it easier for us to solve our real world problems by thinking solution of the problem in terms of real world objects.

So we can say that in our daily life everything can be taken as an object that behaves in a certain way and has certain attributes.

In object orientation we move our concentration to objects in contrast to procedural paradigm in which we simply write our code in functions and call them in our main program.

01.2.What is a Model?

A model is an abstraction of something real or conceptual.
We need models to understand an aspect of reality.

© Virtual University of Pakistan

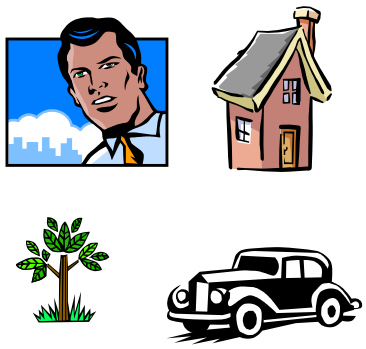
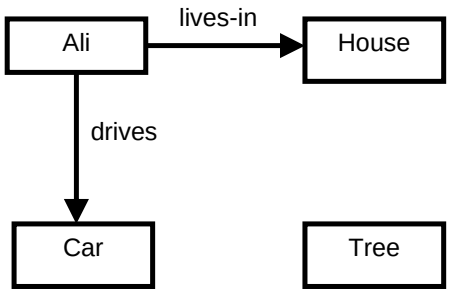
Model Examples

Highway maps
Architectural models
Mechanical models

01.3.OO Models:

In the context of programming models are used to understand the problem before starting developing it.
We make Object Oriented models showing several interacting objects to understand a system given to us for implementation.

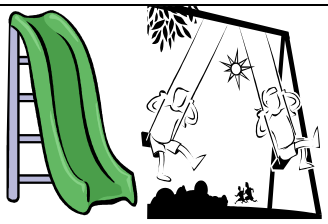
Example 1– Object Oriented Model

<table>
<tr>
<td></td>
<td>
</td>
</tr>
<tr>
<td>Objects
Ali, Car, House, Tree</td>
<td>Interactions
Ali lives in the house
Ali drives the car</td>
</tr>
</table>

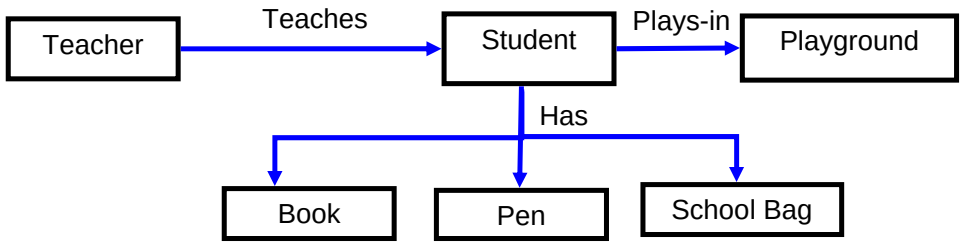
Example 2– Object Oriented Model (A School Model)







A School Model



Objects	**Interactions**
Teacher, Student, School Bag, Pen, Book Playground	Teacher teaches Student. Student has School Bag, Book and Pen

01.4.Object-Orientation - Advantages

As Object Oriented Models map directly to reality as we have seen in examples above therefore,

We can easily **develop** an object oriented model for a problem.
Everyone can easily **understand** an object oriented model.
We can easily implement an object oriented model for a problem using any object oriented language like c++ using its features[1] like classes, inheritance, virtual functions and so on…

01.5.What is an Object?

An object is,

1. Something tangible (Ali, School, House, Car).
2. Something conceptual (that can be apprehended intellectually for example time, date and so on…).

An object has,

1. State (attributes)
2. Well-defined behavior (operations)

[1] We will study these features in detail in this course

© Virtual University of Pakistan

3. Unique identity

01.6.Tangible and Intangible Objects

Examples of Tangible Objects:

Ali is a tangible object, having some characteristics (attributes) and behavior as given below,

Ali	
Characteristics (attributes)	**Behaviour (operations)**
Name Age	Walks Eats

We will identify Ali using his name.

Car is also a tangible object having some characteristics (attributes) and behavior given below,

Car	
State (attributes)	**Behavior (operations)**
Color Model	Accelerate Start Car Change Gear

We can identify Car using its registration number

Examples of Intangible Objects (also called as conceptual objects):

Time is an intangible (conceptual) object

Time	
State (attributes)	**Behavior (operations)**
Hours Seconds Minutes	Set/Get Hours Set/Get Seconds Set/Get Minutes

We will assign our own generated unique ID in the model for Time object

Date is also an intangible (conceptual) object

State (attributes)	**Behavior (operations)**
Year Day Month	Set/Get Year Set/Get Day Set/Get Month

We will assign our own generated unique ID in the model for Date object.

01.7.Summary:

- Model is the abstraction of some real word scenario. It helps us to understand that scenario.
- Object oriented model of any scenario (problem) describes that scenario (problem) in the form of interacting objects.
- We use Object Orientation because it helps us in mapping real world problem in a programming language.
- Object Orientation is achieved using objects and their relationships.
- Properties of an object are described using its *data members* and behavior of an object is described using its *functions*.
- Objects may be tangible (physical) or intangible (also called conceptual or virtual).
- Generally when we have given a certain problem description, **nouns** in that problem description are *candidates* for becoming objects of our system.
- There may be more than one aspects of an object
- It is not necessary that every object has a specific role in implementation of a problem there may be some objects without any role, like school parking in our school.
- It is easier to develop programs using Object Oriented Programming because it is closer to real life.

© Virtual University of Pakistan

Lecture No.02

Lecture Contents

1. Information Hiding
2. Encapsulation
3. Interface
4. Implementation
5. Separation of Interface & Implementation
6. Messages

02.1.Information Hiding:

Information hiding is one of the most important principles of OOP inspired from real life which says that all information should not be accessible to all persons. Private information should only be accessible to its owner.
By Information Hiding we mean "*Showing only those details to the outside world which are necessary for the outside world and hiding all other details from the outside world.*"

Real Life Examples of Information Hiding

1. Ali's name and other personal information is stored in his brain we can't access this information directly. For getting this information we need to ask Ali about it and it will be up to Ali how much details he would like to share with us.

2. An email server may have account information of millions of people but it will share only our account information with us if we request it to send anyone else accounts information our request will be refused.

3. A phone SIM card may store several phone numbers but we can't read the numbers directly from the SIM card rather **phone-set** reads this information for us and if the owner of this phone has not allowed others to see the numbers saved in this phone we will not be able to see those phone numbers using phone.

In object oriented programming approach we have objects with their attributes and behaviors that are hidden from other classes, so we can say that object oriented programming *follows the principle of information hiding.*

In the perspective of **Object Oriented Programming** Information Hiding is,

"Hiding the object details (state and behavior) from the users"

Here by users we mean **"an object"** of another class that is calling functions of this class using the reference of this class object or it may be some other program in which we are using this class.

Information Hiding is achieved in Object Oriented Programming using the following principles,

- All information related to an object is stored within the object
- It is hidden from the outside world
- It can only be manipulated by the object itself

Advantages of Information Hiding

Following are two major advantages of information hiding,

It simplifies our Object Oriented Model:

As we saw earlier that our object oriented model only had objects and their interactions hiding implementation details so it makes it easier for everyone to understand our object oriented model.

It is a barrier against change propagation

As implementation of functions is limited to our class and we have only given the name of functions to user along with description of parameters so if we change implementation of function it doesn't affect the object oriented model.

We can achieve information hiding using **Encapsulation** and **Abstraction**, so we see these two concepts in detail now,

02.2. Encapsulation

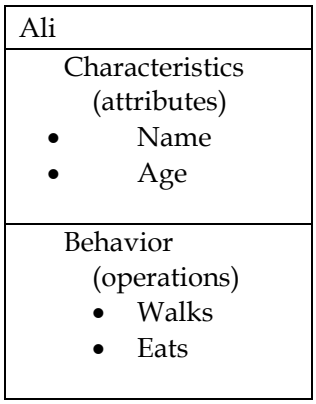
Encapsulation means *"we have enclosed all the characteristics of an object in the object itself"*
Encapsulation and information hiding are much related concepts (information hiding is achieved using Encapsulation)
We have seen in previous lecture that object characteristics include data members and behavior of the object in the form of functions.

So we can say that Data and Behavior are tightly coupled inside an object and both the information structure and implementation details of its operations are hidden from the outer world.

Examples of Encapsulation

Consider the same example of object Ali of previous lecture we described it as follows,

© Virtual University of Pakistan



You can see that Ali stores his personal information in itself and its behavior is also implemented in it.
Now it is up to object Ali whether he wants to share that information with outside world or not. Same thing stands for its behavior if some other object in real life wants to use his behavior of walking it can not use it without the permission of Ali.

So we say that attributes and behavior of Ali are encapsulated in it.

Any other object don't know about these things unless Ali share this information with that object through an interface,

Same concept also applies to phone which has some data and behavior of showing that data to user we can only access the information stored in the phone if phone interface allow us to do so.

Advantages of Encapsulation
The following are the main advantages of Encapsulation,

a. Simplicity and clarity
As all data and functions are stored in the objects so there is no data or function around in program that is not part of any object and is this way it becomes very easy to understand the purpose of each data member and function in an object.

b. Low complexity
As data members and functions are hidden in objects and each object has a specific behavior so there is less complexity in code there will be no such situations that a functions is using some other function and that functions is using some other function.

c. Better understanding
Everyone will be able to understand whole scenario by simple looking into object diagrams without any issue as each object has specific role and specific relation with other objects.

02.3.Interface

Interface is a set of functions of an object that he wants to expose to other objects.

As we discussed previously that data and behavior of each object is hidden in that object it self so we have to use the concept of interface of the object to expose its behavior to outer word objects.

- Different objects may need different functions of an object so interface of an object may be different for different objects.
- Interfaces are necessary for object communication. Each object provides interface/s (operations) to other objects through these interfaces other objects communicate with this object.

Example – Interface of a Car

- Steer Wheels
- Accelerate
- Change Gear
- Apply Brakes
- Turn Lights On/Off

Example – Interface of a Phone

- Input Number
- Place Call
- Disconnect Call
- Add number to address book
- Remove number
- Update number

02.4.Implementation

It is actual implementation of the behavior of the object in any Object Oriented language.

It has two parts,

- Internal data structures to hold an object state that will be hidden from us it will store values for an object data members.
- Functionality in the form of member functions to provide required behavior.

Examples of Implementation

a. Gear Box in car system

Consider object Gear Box in car system it has a certain structure and functionality. When this object will be implemented it will have two things,

- Physical structure of the gear box
- Functionality implemented in this structure to change gear.

Both these things are part of implementation.

© Virtual University of Pakistan

So it has,

- **Data Structure** in the form of Mechanical structure of gear box
- **Functionality** mechanism to change gear

b. Address Book in a Phone

Similarly take the example of contact details saved in the SIM of a phone,

In that case we can say physical structure of SIM card as **Data Structure** And Read/write operations provided by the phone as **Functionality**.

02.5.Separation of Interface & Implementation

As discussed earlier we only show interface of an object to outside world and hide actual implementation from outside world. The benefit of using this approach is that our object interface to outside word becomes independent from inside implementation of that interface.

This is achieved through the concepts of encapsulation and information hiding.

Real Life example of separation of interface and implementations

- Driver has a standard interface to drive a car and using that interface he drive can drive any car regardless of its model or type whatever engine type it has or whatever type of fuel it is using.

02.6.Messages

Objects communicate through messages they send messages (stimuli) by invoking appropriate operations on the target object. The number and kind of messages that can be sent to an object depends upon its interface

Examples – Messages

A Person sends message (stimulus) "stop" to a Car by applying brakes

A Person sends message "place call" to a Phone by pressing appropriate button

02.7.Summary

- Information hiding is achieved through encapsulation.
- Encapsulation and Information Hiding are related to each other.
- Interface of an object provides us the list of available functions.
- An object may have more than one interface.
- Interface and implementation are separated from each other to achieve Information Hiding.
- Objects communicate with each other using messages.

Useful Links:

http://www.alice.org/

A Graphical Programming Environment to teach Computer Programming.

© Virtual University of Pakistan

Lecture No.03

Lecture Contents:

- Abstraction
- Classes
- Inheritance
- Major benefits of inheritance (Reuse)

03.1. Abstraction

Real life objects have a lot of attributes and many kind of behaviors but most of the time we are interested in only that part of the objects that is related to the problem we are currently going to solve, for example in implementing a school system we don't need to take care of the personnel life of a student or a teacher as it will not effect our system in any way so we will see these objects in the perspective of school system and will ignore their other characteristics, this concept is called "*Abstraction*". Abstraction is a way to cope with complexity and it is used to simplify things.

Principle of abstraction:

"Capture only those details about an object that are relevant to current perspective"

Abstraction Example:

Suppose we want to implement abstraction for the following statement,

"Ali is a PhD student and teaches BS students"

*Here object Ali has two **perspectives** one is his **student perspective** and second is his **teacher perspective.***

We can sum up Ali's attributes as follows,

Name
Age
Student Roll No
Year of Study
CGPA
Employee ID
Designation
Salary

As you can see out of all these listed attributes some belong to Ali's student perspective(Roll No, CGPA, Year of study) and some belong to Ali's teacher perspective(Employee ID, Designation, Salary).

Similarly we can sum up Ali's behavior as follows,

Study
DevelopExam

GiveExam
TakeExam
PlaySports
Eat
DeliverLecture
Walk

As was the case with attributes of object Ali, its behavior can also be divided in Ali's student perspective as well as Ali's teacher perspective.

Student's Perspective

Attributes:

- **Name**
- **Student Roll No**
- **Year of Study**
- **CGPA**
- Employee ID
- Designation
- Salary
- Age

Behaviour:

- **Study**
- **GiveExam**
- **PlaySports**
- DeliverLecture
- DevelopExam
- TakeExam
- Eat
- Walk

Teacher's Perspective

Attributes:

- **Name**
- Student Roll No
- Year of Study
- CGPA
- **Employee ID**
- **Designation**
- **Salary**
- Age

Behaviour:

- Study
- GiveExam
- PlaySports
- **DeliverLecture**
- **DevelopExam**
- **TakeExam**
- Eat
- Walk

A cat can be viewed with different perspectives

Ordinary Perspective	Surgeon's Perspective
A pet animal with Four Legs A Tail	A being with A Skeleton Heart

© Virtual University of Pakistan

Two Ears Sharp Teeth	Kidney Stomach

A car can be viewed with different perspectives

 Driver's View	 Engineer's View

Abstraction – Advantages

Abstraction has following major advantages,

1. It helps us understanding and solving a problem using object oriented approach as it hides extra irrelevant details of objects.

2. Focusing on single perspective of an object provides us freedom to change implementation for other aspects of for an object later.

Similar to Encapsulation Abstraction is also used for achieving information hiding as we show only relevant details to related objects, and hide other details.

03.2.Classes

In OOP we create a general sketch for each kind of objects and then we create different instances using this sketch we call this sketch or prototype or map as "class".
All objects of same kind exhibit identical characteristics (information structure and behavior) however they have data of their own.

Class –Example 1

Consider the objects given below,

- Ali studies mathematics
- Anam studies physics
- Sohail studies chemistry

Each one is a Student so we say these objects are *instances* of the Student class.

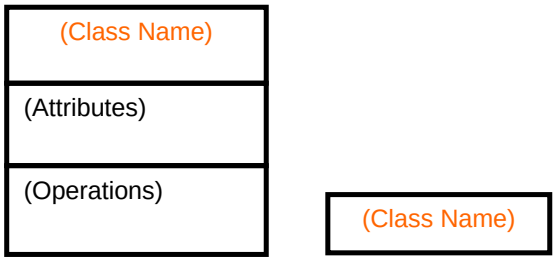
Class –Example 2

Consider the objects given below,

- Ahsan teaches mathematics
- Aamir teaches computer science
- Atif teaches physics

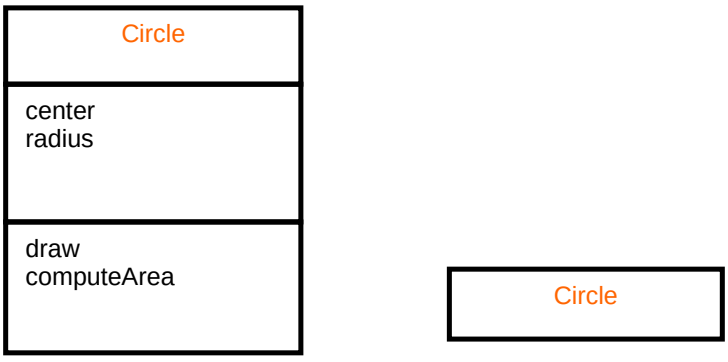
Each one is a teacher so we say these objects are *instances* of the Teacher class

Class Representation:
we can represent a class using a rectangle as follows,



Normal Form **Suppressed Form**

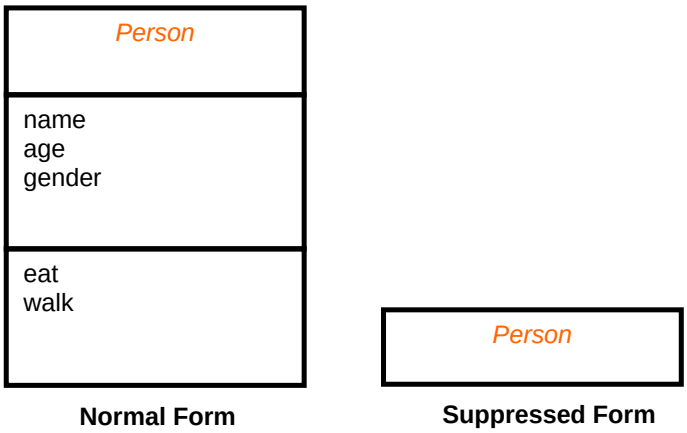
Class Example: Circle



Normal Form **Suppressed Form**

Class Example: Person

© Virtual University of Pakistan

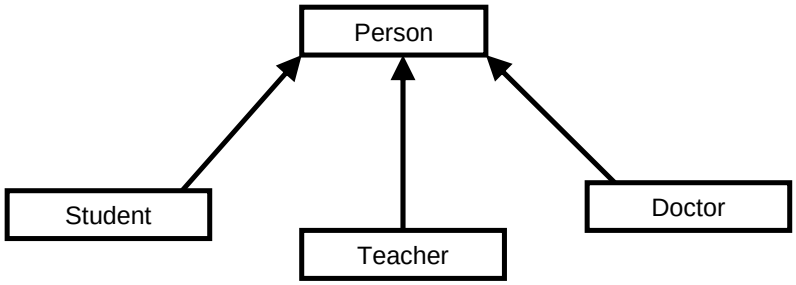


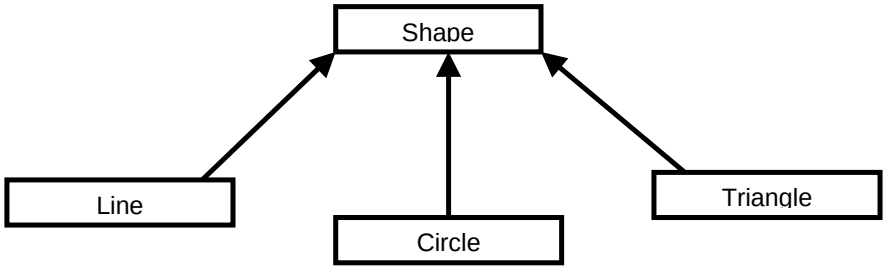
03.3.Inheritance

A child inherits characteristics of its parents, besides inherited characteristics, a child may have its own unique characteristics

Inheritance in Classes

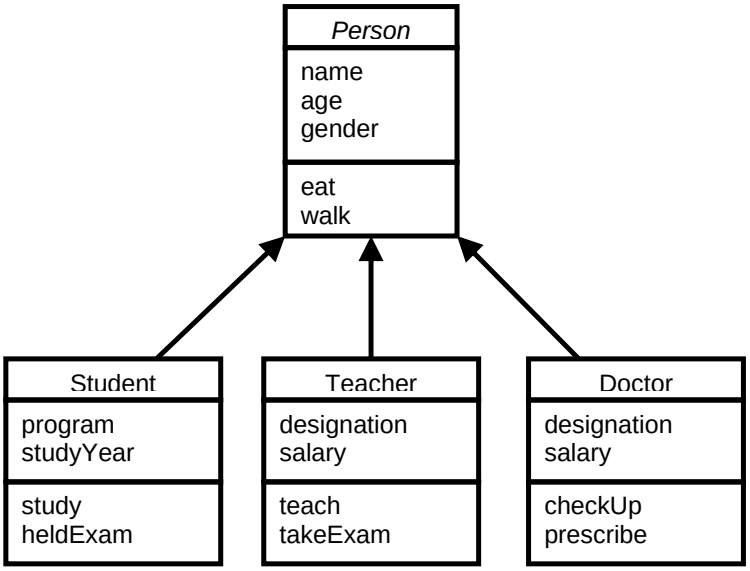
If a class B inherits from class A then it contains all the characteristics (information structure and behaviour) of class A
The parent class is called *base* class and the child class is called *derived* class
Besides inherited characteristics, derived class may have its own unique characteristics



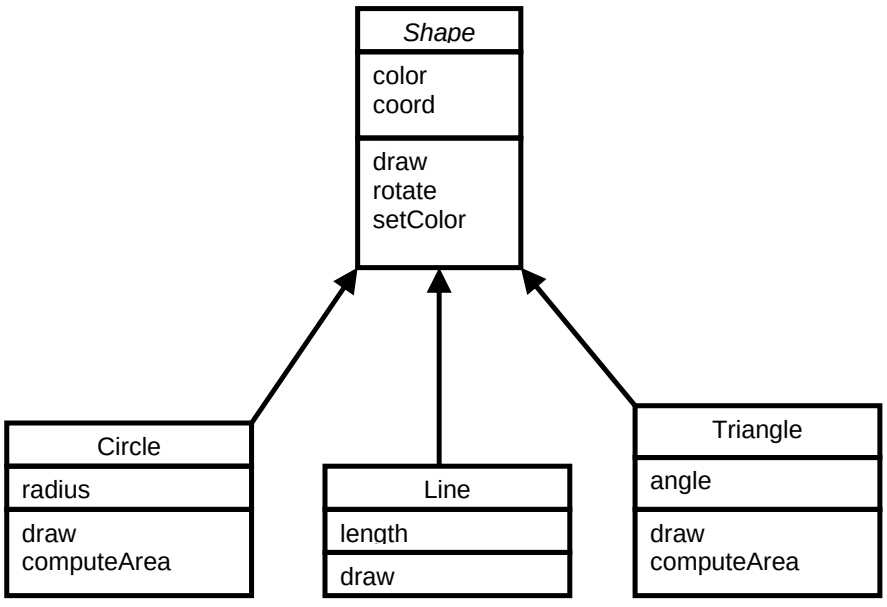


Inheritance – "IS A" or "IS A KIND OF" Relationship

Each derived class is a kind of its base class



Here,
Student IS A Person
Teacher IS A Person
Doctor IS A Person



Here,
Circle IS A Shape
Line IS A Shape
Triangle IS A Shape

© Virtual University of Pakistan

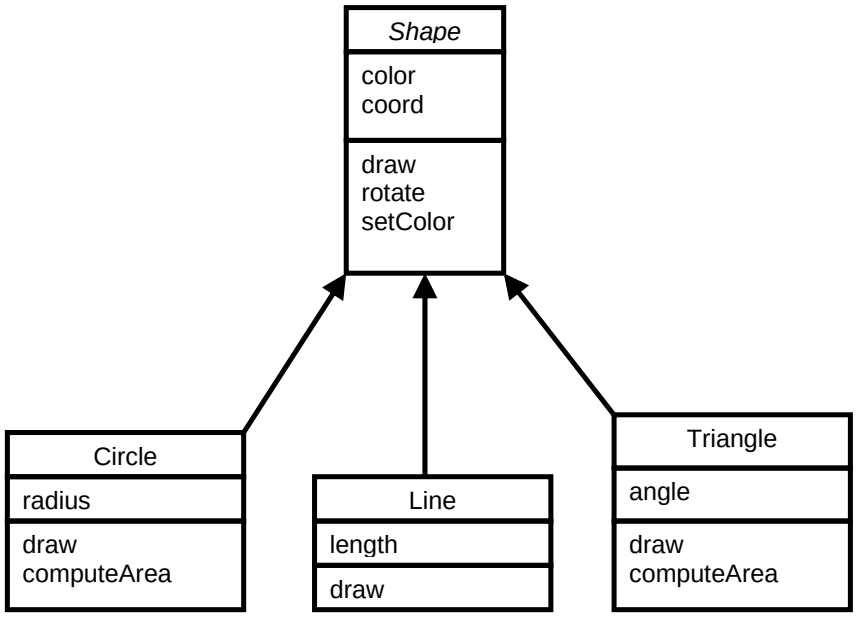
Inheritance – Advantages

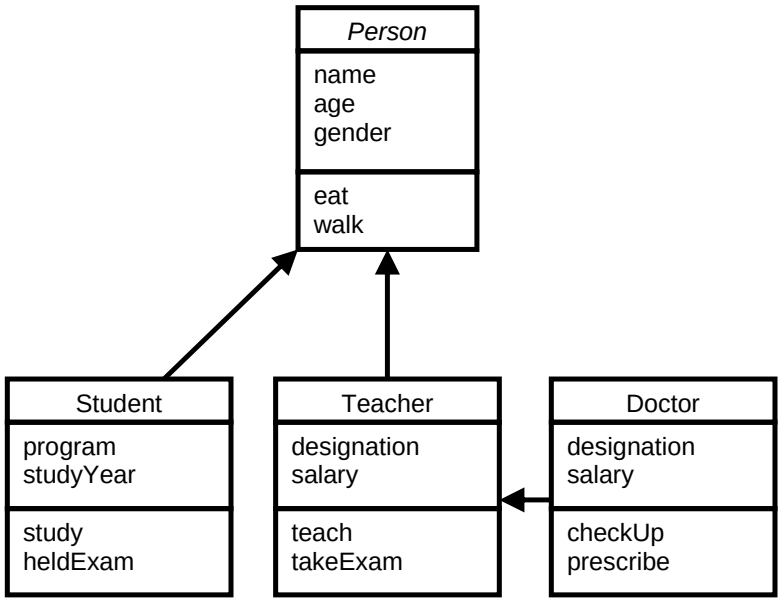
1. Reuse
2. Less redundancy
3. Increased maintainability

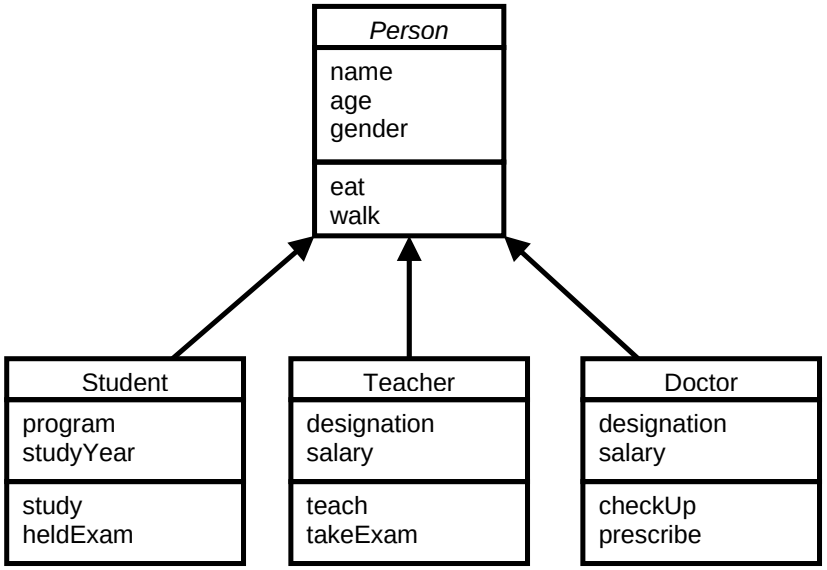
Reuse with Inheritance

Main purpose of inheritance is reuse, we can easily add new classes by inheriting from existing classes.

Select an existing class closer to the desired functionality, create a new class and inherit it from the selected class, add to and/or modify the inherited functionality







 © Virtual University of Pakistan

Lecture No.04

Lecture Contents

- Generalization
- Sub typing (extension)
- Specialization (restriction)
- Overriding
- Abstract classes
- Concrete classes

Recap – Inheritance

- Derived class inherits all the characteristics of the base class
- Besides inherited characteristics, derived class may have its own unique characteristics
- Major benefit of inheritance is reuse

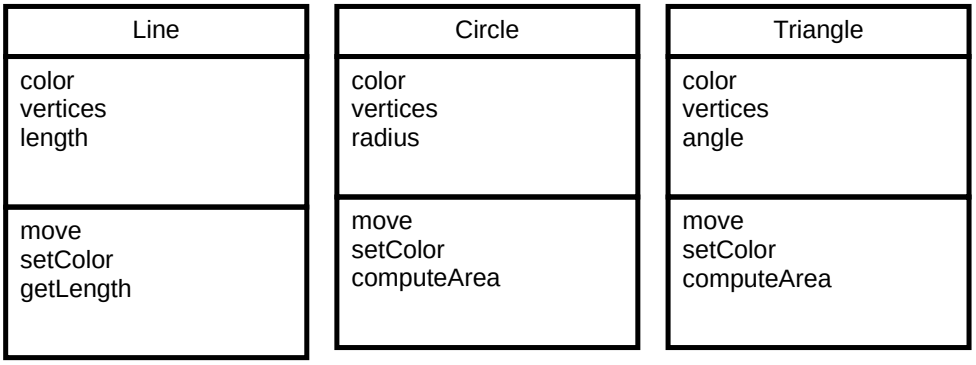
04.1. Concepts Related with Inheritance

- Generalization
- Subtyping (extension)
- Specialization (restriction)

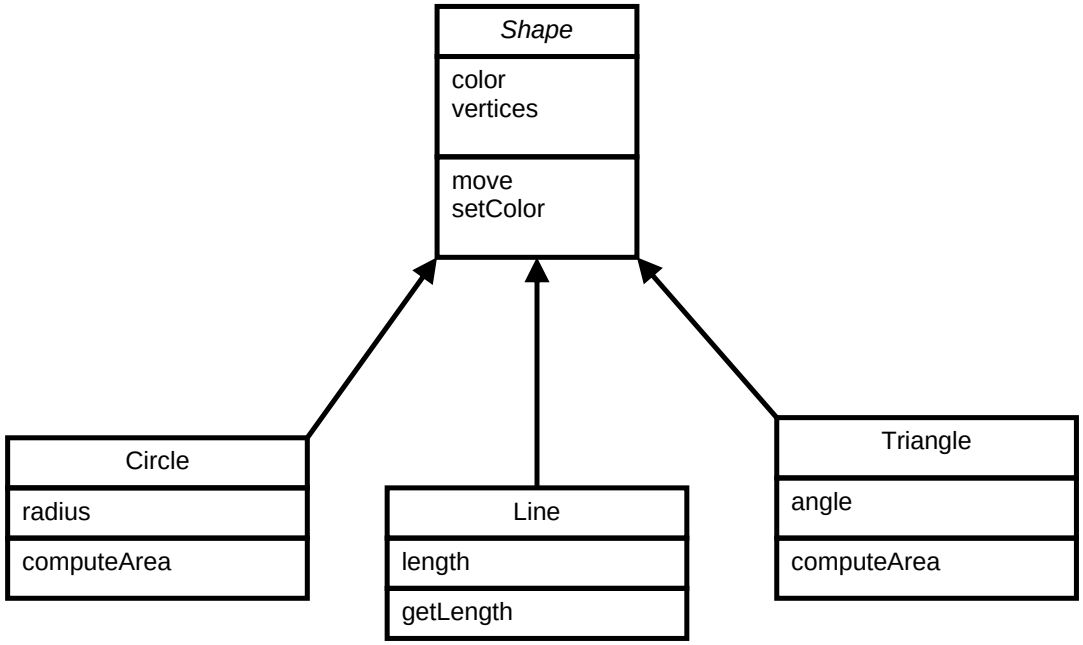
04.2. Generalization

In OO models, some classes may have common characteristics.
We extract these features into a new class and inherit original classes from this new class. There are many objects with common characteristics in object model. The common characteristics (attributes and behaviour) of all these objects are combined in a single general class. Base class encapsulates the idea of commonality of derived classes. Base class is general class representing common behaviour of all derived classes.
This concept is known as Generalization.
It reduces the redundancy and gives us reusability, using generalization our solution becomes less complex.
In generalization there should be "Is a Kind of Relationship" (also called "Is A relationship") between base and child classes.

Example: Line, Circle and Triangle



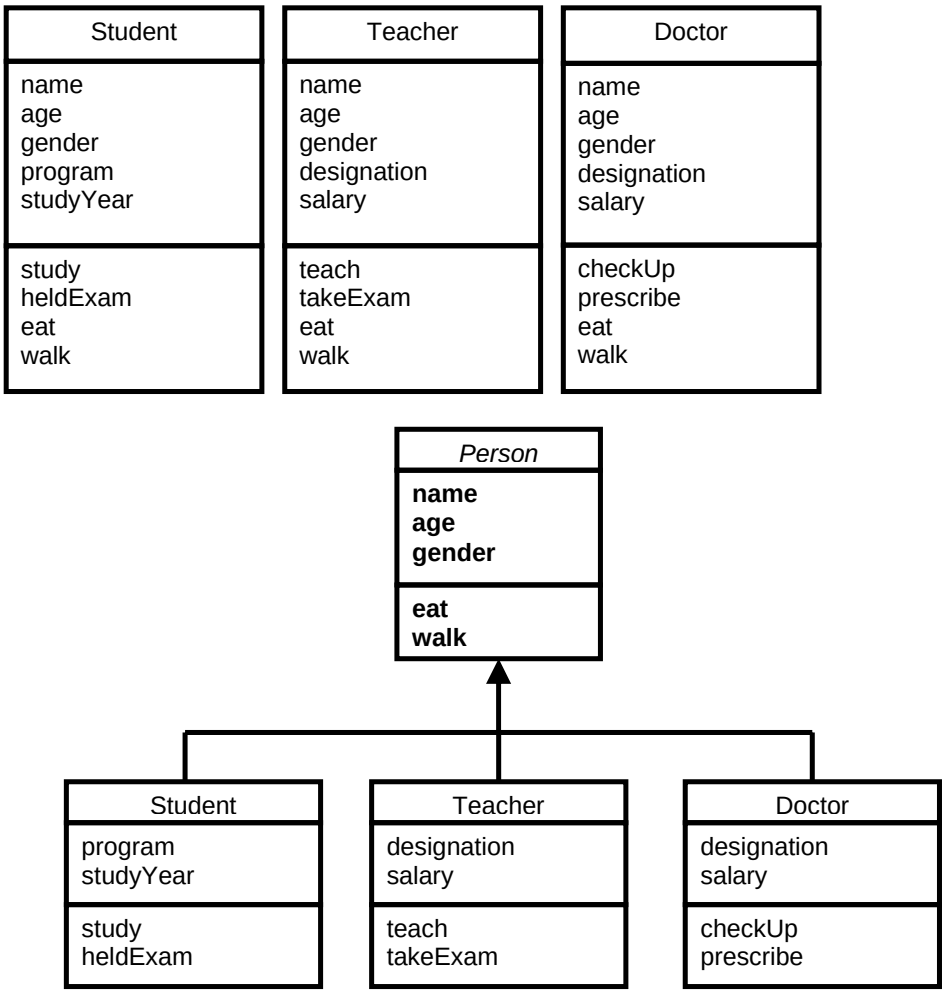
Line is shape Circle is a shape Triangle is a shape



Common attributes
Color vertices
Common behaviour
Set Color, Move

Example: Student Doctor and Teacher

 © Virtual University of Pakistan



Common attributes,
Name, age, gender

Common behaviour
Eat, Walk

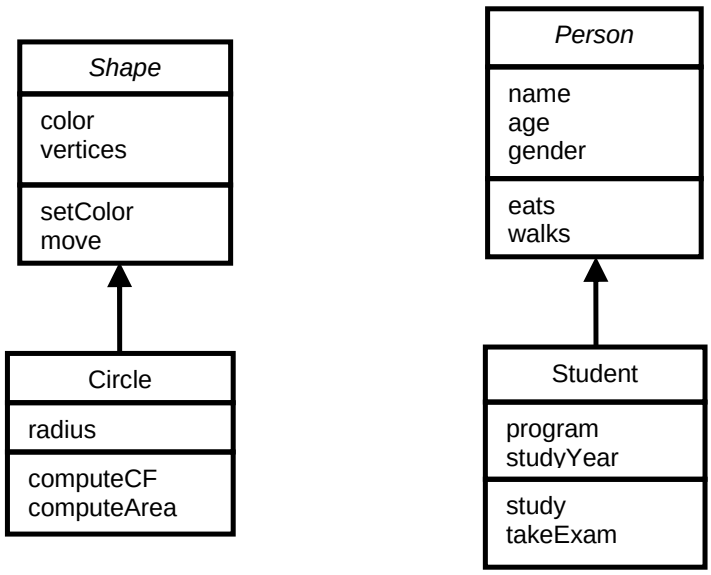
Sub-typing & Specialization

We want to add a new class to an existing model
We have developed an existing class hierarchy
Find an existing class that already implements some of the desired state and behaviour
Inherit the new class from this class and add unique behaviour to the new class

04.3.Sub-typing (Extension)

Sub-typing means that derived class is behaviourally compatible with the base class
Derived class has all the characteristics of base class plus some extra characteristics
Behaviourally compatible means that base class can be replaced by the derived class

Sub-typing (Extension) - Example



Circle is extending the behaviour of shape, it is extending attributes of shape by adding radius similarly it is extending behaviour of shape by adding compute Circumference and compute Area.

Student has two extra attributes program and studyYear
Similarly it has extended behaviour by adding study and takeExam.

Subtyping and generalization are related concepts, Subtyping (extension) and generalization is a way to look same thing in two ways.
Sub typing is looking at things from Top to bottom whereas in generalization we look at things from bottom to top.

04.4.Specialization (Restriction)

We want to add a class to existing hierarchy of classes having many similarities to already existing classes but some part of its behaviour is different or restricted. In that case we will use the concept of specialization.

Specialization means that derived class is behaviourally incompatible with the base class

Behaviourally incompatibility means that base class can't always be replaced by the derived class

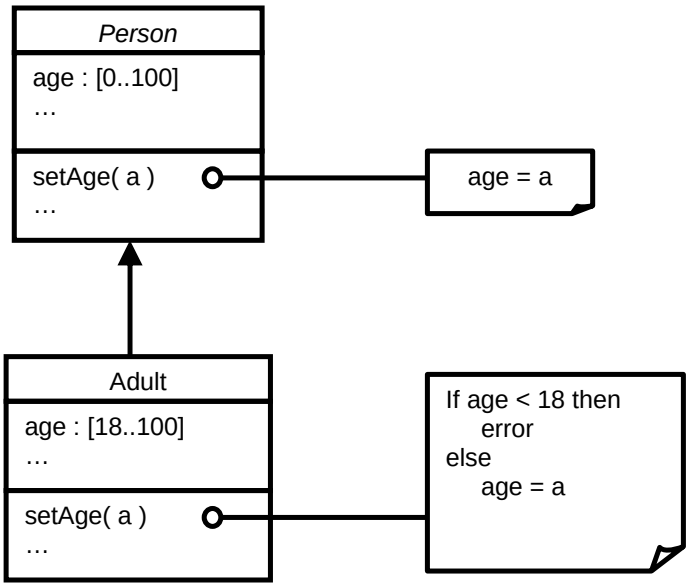
Derived class has some different of restricted characteristics than of base class.

Example – Specialization (Restriction)

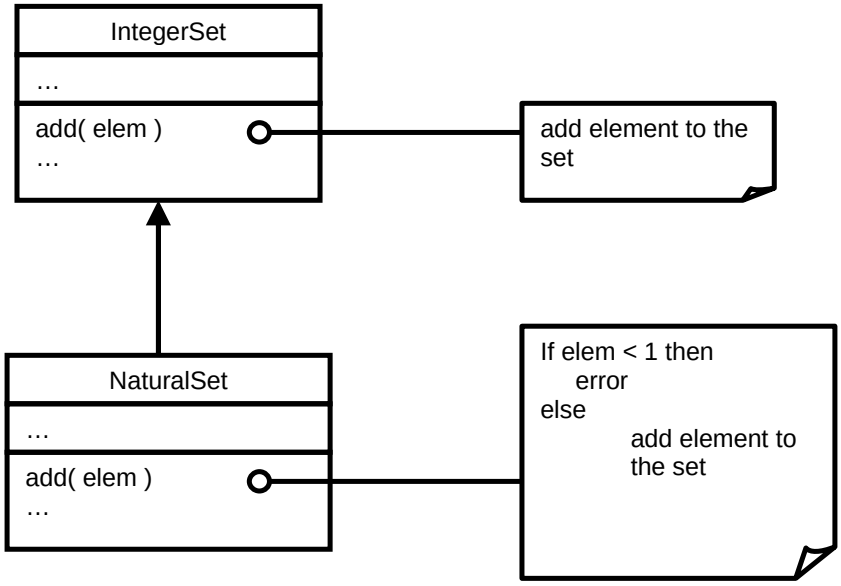
Suppose we want to add one more class of Adult for some special requirement like for ID card generation such that it is a person but its age is greater than 18 and having all other behaviour of that of person class. One solution is that we write

 © Virtual University of Pakistan

another class from beginning and write all code of person again in it with age limit, but better solution is that we derive adult class from person class and restrict age in that class as shown below in diagram,



Similarly Natural Numbers[2] are also Integers[3] with the restriction that natural numbers set can NOT contain zero or negative integers it consists of only positive integers so we can implement this relationship also as specialization,



[2] Natural numbers: positive integers only (numbers from 1 to …….onwards)

[3] Integers: all positive and negative numbers (…..-3 , -2 , -1 , 0 , 1 , 2 , 3……)

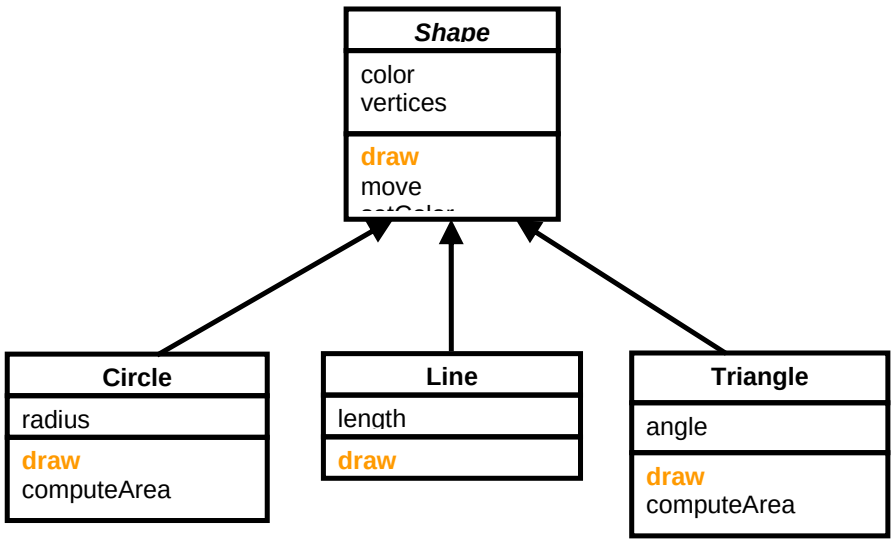
Add method behaviour is present in both base and derived classes but derived class behaviour is different in derived class. Derived class will not exhibit the behaviour of base class but it is **overriding** behaviour of base class with its own behaviour.

04.5.Overriding

A class may need to override the default behaviour provided by its base class
Derived class overrides the behaviour of its base class.
Reasons for overriding
- Provide behaviour specific to a derived class (specialization)
- Extend the default behaviour (extension)
- Restrict the default behaviour (restriction)
- Improve performance

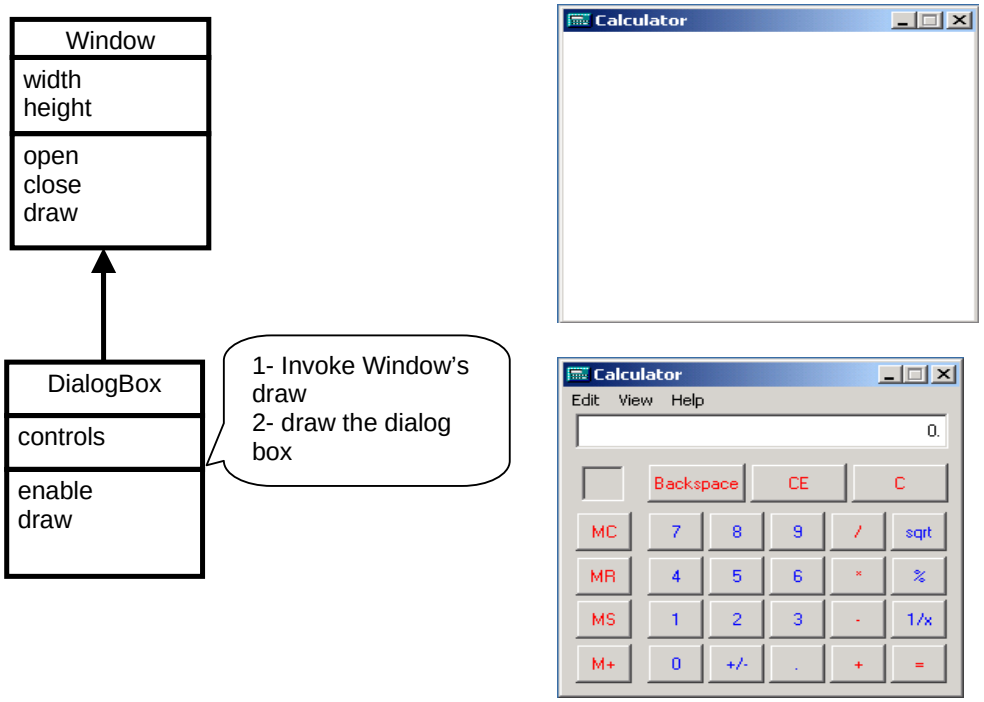
It is used for the implementation of inheritance.

Example – Specific Behaviour (Specialization)

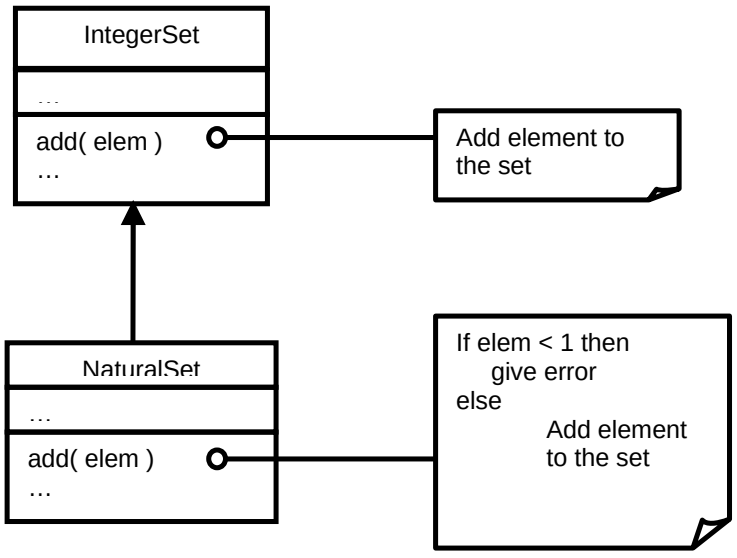


© Virtual University of Pakistan

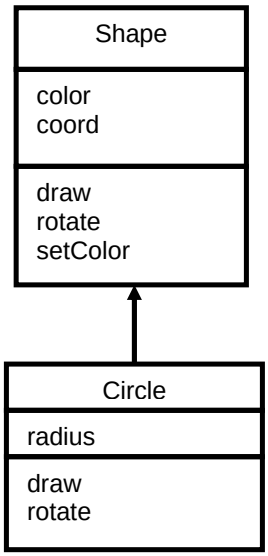
Example – Extention



Example – Restriction



Example – Improve Performance
Class Circle overrides *rotate* operation of class Shape with a Null operation.

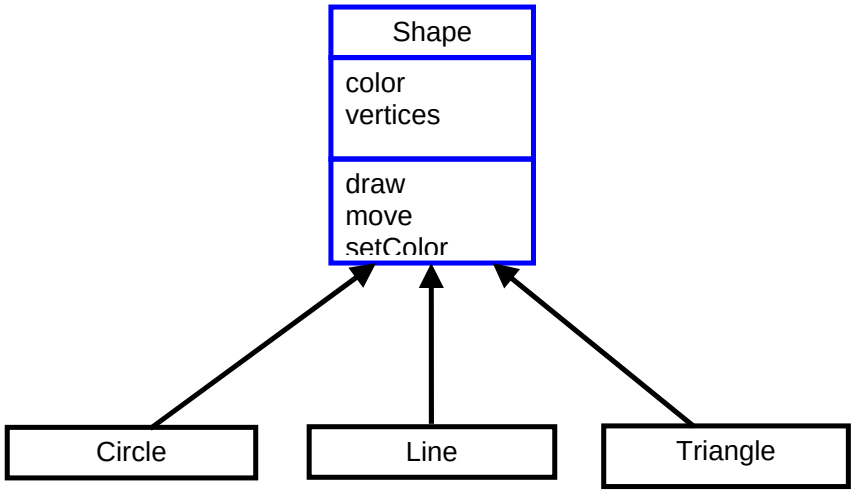


04.6. Abstract Classes

In our examples we made classes for shape and person. These are abstract concepts and the classes we make against abstract concepts are called abstract classes. They are present at or near the top in the class hierarchy to present most generalized behaviour.

An abstract class implements an abstract concept
Main purpose is to be inherited by other classes
Can't be instantiated
Promotes reuse

Abstract Classes - Example I

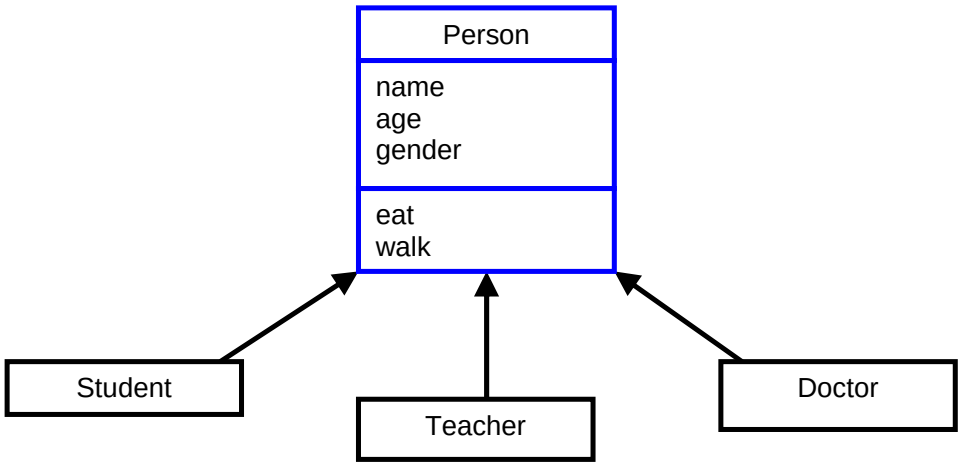


Here, Shape is an abstract class

© Virtual University of Pakistan

Abstract Class	Shape			
Concrete Classes	Circle	Line	Triangle	….

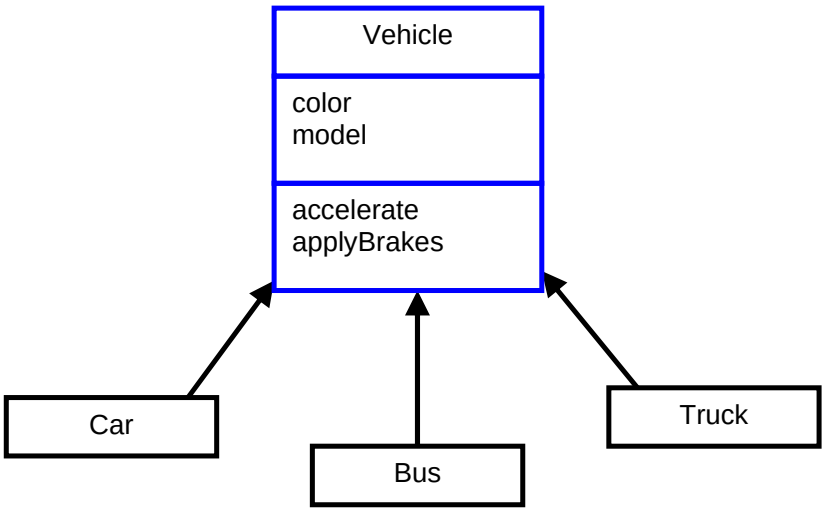
Abstract Classes - Example II



Here, Person is an abstract class

Abstract Class	Person					
Concrete Classes	Student	Teacher	Doctor	Engineer	Director	….

Abstract Classes - Example III



Here, Vehicle is an abstract class

Abstract Class	Vehicle			
Concrete Classes	Car	Bus	Truck	….

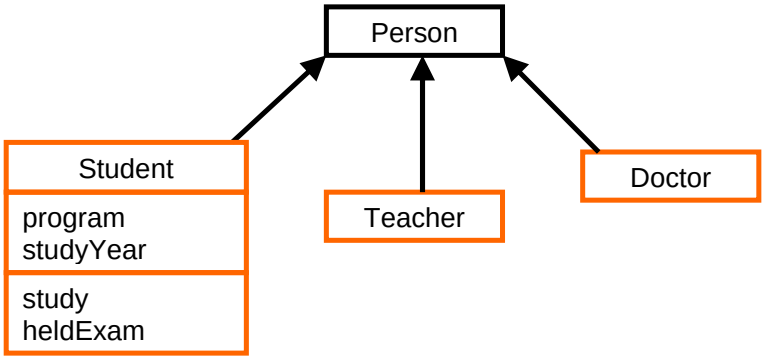
Abstract Classes can not exist standalone in an object model
While making object model we start by finding out objects in our object model and then we find out objects having common attributes and make them in the form of general classes at the top of class hierarchies.

04.7.Concrete Classes

The entities that actually we see in our real world are called concrete objects and classes made against these objects are called concrete classes.

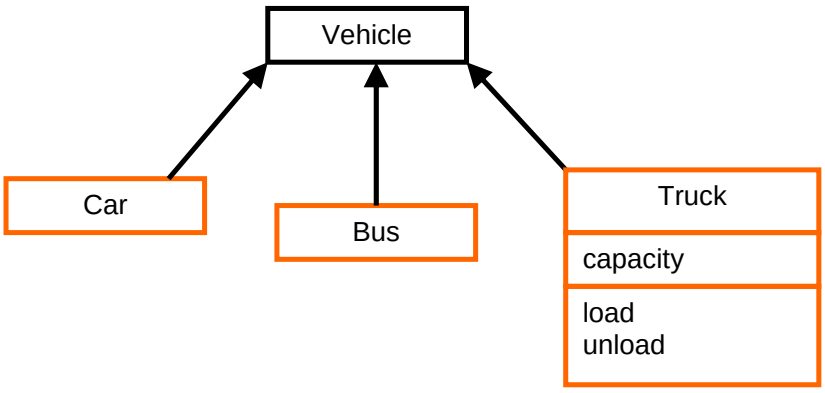
A concrete class implements a concrete concept
These are used to instantiate objects in our programs
Provides implementation details specific to the domain context

Concrete Classes - Example I



Here Student, Teacher and Doctor are concrete classes

Concrete Classes - Example II



Here Car, Bus and Truck are concrete classes

© Virtual University of Pakistan

- A concrete class may exist in an object model independently
- Concrete classes mostly lie below the top of class hierarchy in a good object model.

If there is an abstract class then hierarchy exists in the object model as there will definitely be some concrete classes as well derived from this abstract class otherwise there is no use of abstract class.

Glossary:

a. Natural numbers: numbers from 1 to …….onwards
b. Integers: all positive and negative numbers …..-3,-2,-1,0,1,2,3……
c. Whole numbers: numbers from 0 ,1 ,2, 3 ….onwards (natural no's including 0)

Some times whole numbers are also called numbers without fractional part.

Lecture No.05
Multiple Inheritance

Inheritance:
We saw inheritance purposes in last lecture

- Generalization
- Extention or sub typing
- Specialization or restriction

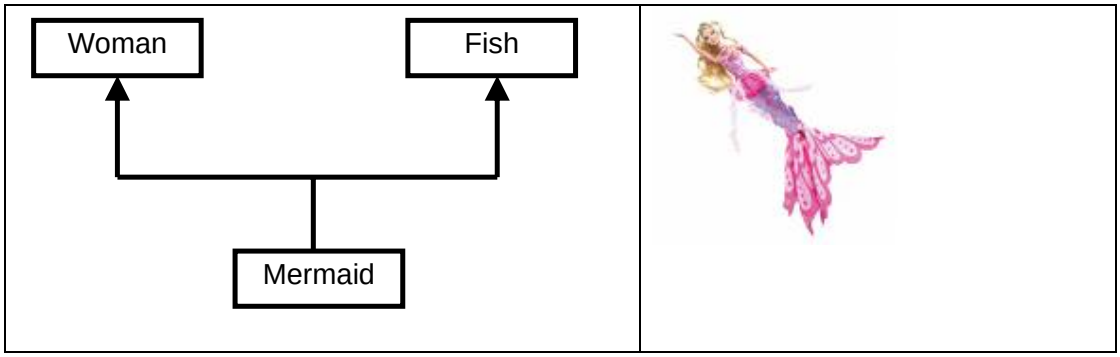
Abstract and concrete classes, former is used to represent abstract concepts later is used to represent concrete concepts.
Overriding derived classes override inherited classes (base classes) behaviour.
Overriding is used for Specialization, Extention, Restriction, and Performance.

05.1.Multiple Inheritance

Sometimes we want to reuse characteristics of more than one parent class, in that case we need to inherit a class from more than one classes.

Example 1– Multiple Inheritance

Consider the example of an imaginary specie Mermaid used in fairy tales that lives in water having features both of a women as well as of a fish, In Object Oriented programming perspective Mermaid can be derived from two classes Women and Fish.



C++ Code:

```
/*Program to demonstrate simple multiple inheritance*/

class Fish {


};

class Woman {
```

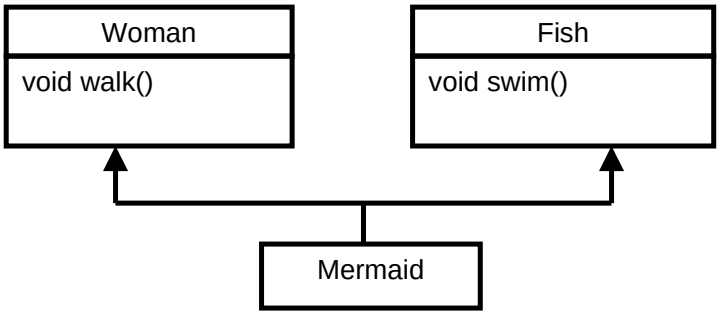

© Virtual University of Pakistan

```
};

class Mermaid : public Woman , public Fish {

};
```

Our Mermaid class inherits features of both woman and fish suppose our woman class has method wald() and fish cclass has method swim then our mermaid class can use both methods i.e can walk as well as can swim.



C++ code:

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

using namespace std;

/*Program to demonstrate simple multiple inheritance*/

class Fish
{
public:
    void swim(){
    cout<<"\n In method swim";
    }

};

class Woman
{
public:
    void walk(){
    cout<<"\n In method walk"<<endl;
    }

};
```

```
class Mermaid : public Woman,public Fish
{

};

int main(int argc, char *argv[])
{

Mermaid mermaid;
/*This Mermaid object will have two implicit objects one of Fish class and one of
Woman class*/
mermaid.swim();
mermaid.walk();

system("PAUSE");
return 0;

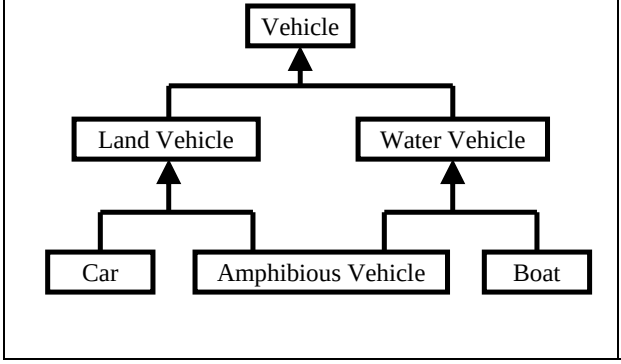
}
```

Output:

```
In method[4] swim
In method walk
```

Example 2– Multiple Inheritance

Take another example of amphibious vehicle (vehicle that can run on land as well as on water) so it has properties of both land as well as of water vehicle. The general hierarchy in this case will be,





Here we have added a general Vehicle class as well to add all common functions of Land Vehicles and Water Vehicles in that class, and specific functions of Land and

[4] class member functions are also called class methods

© Virtual University of Pakistan

Water vehicle in their respective classes then we have derived Amphibious Vehicle class from Land Vehicle and Water Vehicle classes (we can do the same in first example as well concerning Woman, Fish and Mermaid).

C++ code:

```
class Vehicle
{

};

class WaterVehicle : public Vehicle
{


};

class LandVehicle : public Vehicle
{


};

class AmphibiousVehicle : public LandVehicle,public WaterVehicle
{

};
```

Suppose we have a **changeGear** method in Vehicle class that is applicable to both water and land vehicle, we also have Float and Move methods in water and land vehicles respectively then our amphibious vehicle will have all these methods,

C++ code:

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

using namespace std;

/*Multiple Inheritance in case of Amphibious Vehicle*/

class Vehicle
{
public:
 void changeGear(){ cout<<"\nI am Vehicle changeGear() function..\n";}
};

class WaterVehicle : public Vehicle
{
```

```
public:
void Float(){ cout<<"\nI am float function of Water Vehicle";}
};


class LandVehicle : public Vehicle
{
public:
void Move(){ cout<<"\nI am move function of Land Vehicle"<<endl;}

};

class AmphibiousVehicle : public LandVehicle,public WaterVehicle
{


};

int main(int argc, char *argv[])
{

AmphibiousVehicle amphibious;

amphibious.Float();
/*Calling Float function of Water Vehicle class*/

amphibious.Move();
/*Calling Move function of Land Vehicle class*/

system("PAUSE");
return 0;


}
```

Output:

```
I am float function of Water Vehicle
I am move function of Land Vehicle
```

Advantage of Multiple Inheritance:

As was the case with simple (single) inheritance multiple inheritance also decreases redundant code as we can inherit a class from many classes and can use their functions without the need to write them again.

© Virtual University of Pakistan

However, there are more disadvantages of multiple inheritance, than its advantages.

Problems with Multiple Inheritance

Increased complexity

Amphibious vehicle hierarchy is a complicated as this class is derived from two classes that will make code more complex and less understandable however this is obvious as amphibious vehicle is a complicated vehicle. It is generic problem.

Reduced understanding

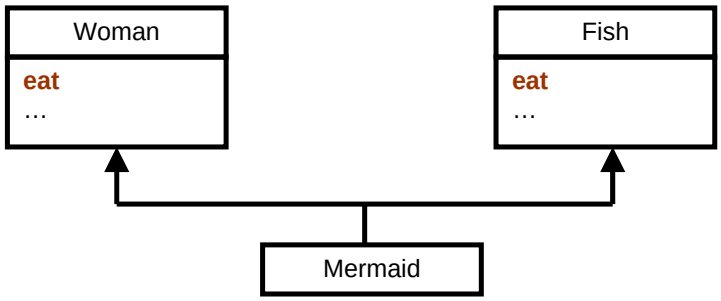
Due to increased complexity of class hierarchy the object model becomes difficult it understand especially for someone who is looking it first time.

Duplicate features

As we are deriving a single class from more than one class so there is a chance of duplication of features (same methods in both parents), following problems may arise due to duplicate features,

Problem 1: Ambiguity

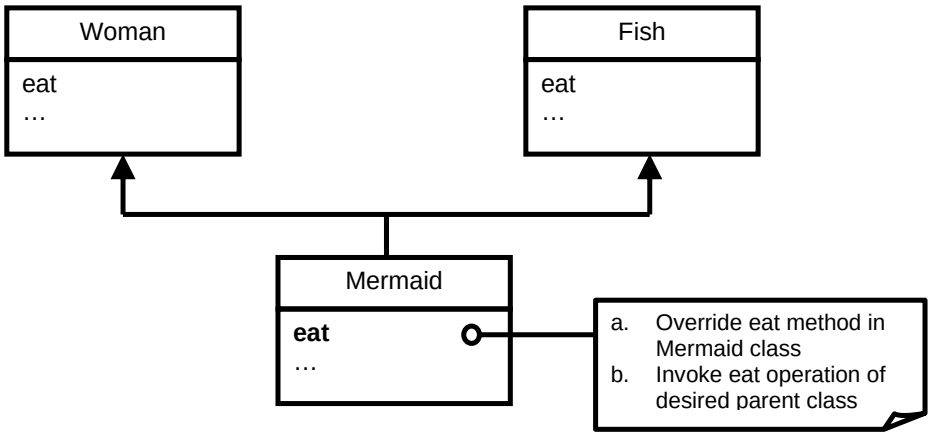
Consider the class hierarchy of Mermaid class below,



As mermaid also needs to eat and its both parents have their own methods of eating so here question arises,

Which eat operation Mermaid should inherit as both functions are available?

Solution – We can solve this problem by explicitly calling eat method from any of the parent classes in Mermaid class according to behaviour of Mermaid (i.e. if it eats like a Woman we can call eat method of Woman class and if eats like Fish we can call method of Fish class), for this we will Override the Common method in multiply inherited class and in that class overridden method we will call the appropriate base class function.



Example C++ Code

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

using namespace std;

/*Program to demonstrate simple multiple inheritance*/

class Fish
{
public:
      void eat(){
      cout<<"\n In Fish eat method ";
      }

};

class Woman
{
public:
      void eat(){
      cout<<"\n In Woman eat method \n"<<endl;
      }

};

class Mermaid : public Woman,public Fish
{

     public:

      void eat(){
```

 © Virtual University of Pakistan

```
    cout<<"\n In Mermaid eat method "<<endl;
    cout<<"\n Explicity calling Woman eat method...."<<endl;

    Woman::eat();

    }

};

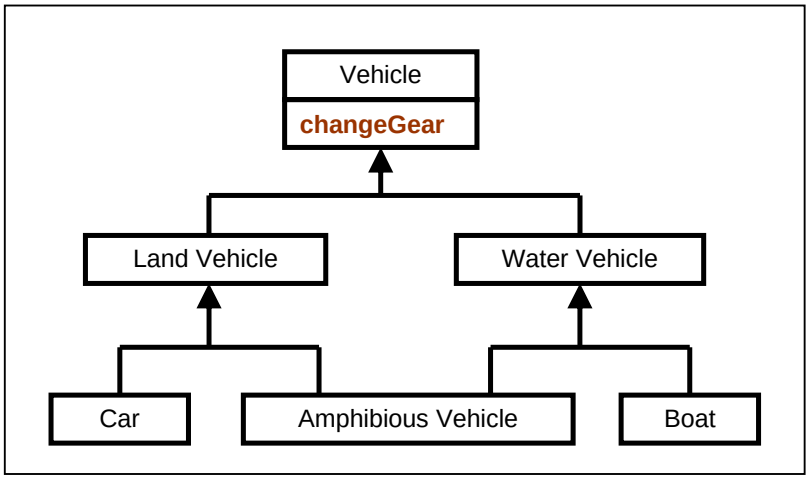
int main(int argc, char *argv[])
{


Mermaid mermaid;
/*This Mermaid object will have two implicit objects one of Fish class and one of
Woman class*/
mermaid.eat();
/*Calling Mermaid eat method*/

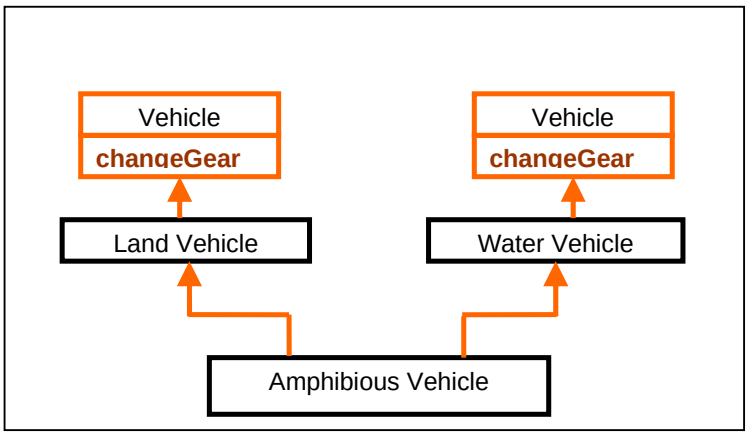
system("PAUSE");
return 0;

}
```

Problem 2: Two instances for same function (Diamond Problem)



Here Amphibious Vehicle will have two copies of changeGear function as it will have two objects of Vehicle class one with respect to Land Vehicle and one with respect to Water Vehicle as shown below,



Actual Memory Layout

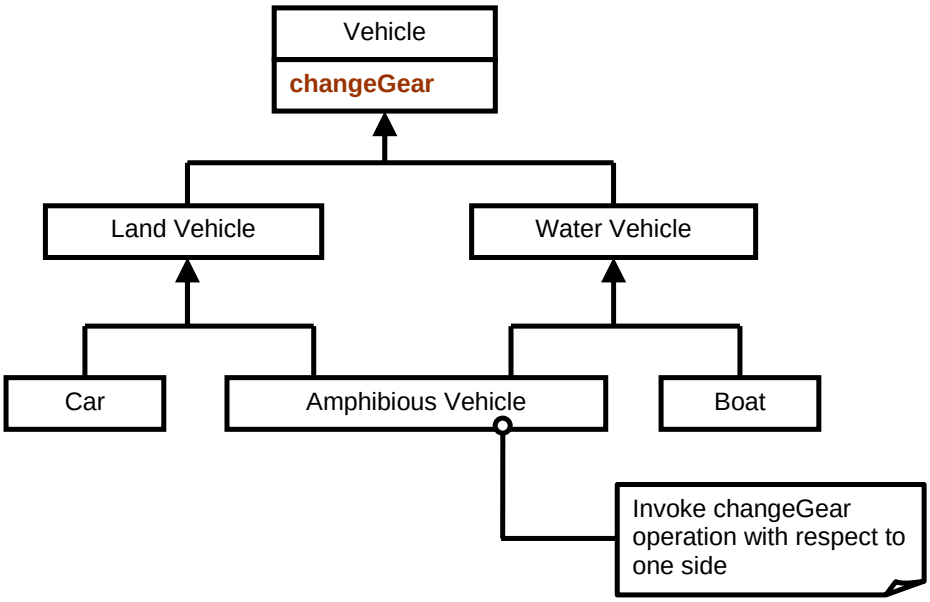
Compiler will not be able to decide which changeGear operation Amphibious Vehicle should inherit so it will generate an error as shown below (two copied of same method),

error: request for member `changeGear' is ambiguous
error: candidates are: **void Vehicle::changeGear()**
void Vehicle::changeGear()
Execution terminated

Solution to Diamond Problem

Some languages disallow diamond hierarchy
Others provide mechanism to ignore characteristics from one side. There are two cases while solving diamond problem virtual inheritance and non virtual inheritance (we will study these details in coming lectures)

© Virtual University of Pakistan



Association:

Interaction of different objects in OO model (or in problem domain) is known as association.
In object oriented model, objects interact with each other in order to perform some useful work, while modeling these objects (entities) is done using the association. Usually an object provides services to several other objects. An object keeps association with other objects to delegate tasks. This association can be represented with a line along an arrow head (⟶) or without arrow head.

05.2. Kinds of Association:
There are two main types of association which are then further subdivided i.e

1. Class Association
2. Object Association

1. **Class Association**

Class association is implemented in terms of Inheritance. Inheritance implements generalization/specialization relationship between objects. Inheritance is considered class association.

- In case of public inheritance it is "IS-A" relationship.
- In case of private inheritance it is "Implemented in terms of" relationship.

This relationship ensures that public members of base class are available to derived class in case of public inheritance.

When we create objects of classes in which we have implemented inheritance relationships we are forcing the inheritance relationship between created objects. In this case the derived class objects will also contain base class objects attributes and methods.

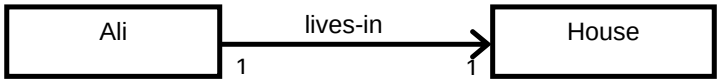
2. **Object Association**

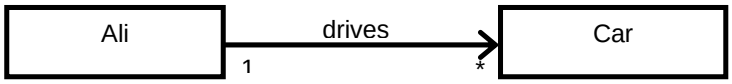
It is the interaction of stand alone objects of one class with other objects of anther class.
It can be of one of the following types,

- Simple Association
- Composition
- Aggregation

05.3.Simple Association

The two interacting objects have no intrinsic relationship with other object. It is the weakest link between objects. It is a reference by which one object can interact with some other object.
Customer gets cash from cashier
Employee works for a company
Ali lives in a house
Ali drives a car





It is generally called as "**association**" instead of "**simple association**"

Kinds of Simple Association

Simple association can be categorized in two ways,

- With respect to direction (navigation)
- With respect to number of objects (cardinality)

Kinds of Simple Association w.r.t Navigation

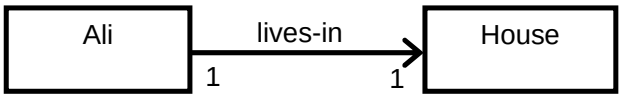
With respect to navigation association has the following types,

a. One-way Association
b. Two-way Association

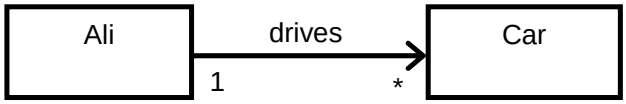
a. One-way Association

In One way association we can navigate along a single direction only, it is denoted by an arrow towards the server object.
Examples:

© Virtual University of Pakistan



- Ali lives in a House

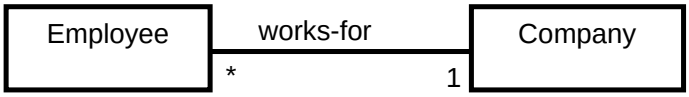


- Ali drives his Car

b. Two-way Association

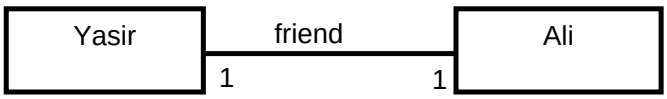
In two way association we can navigate in both directions, it is denoted by a line between the associated objects

Examples:



Employee works for company
Company employs employees

Two-way Association - Example



Yasir is a friend of Ali
Ali is a friend of Yasir

Kinds of Simple Association w.r.t Cardinality

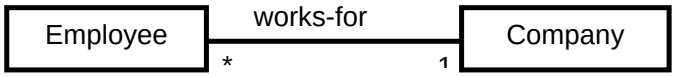
With respect to cardinality association has the following types,

a. Binary Association
b. Ternary Association
c. N-ary Association

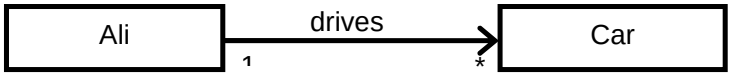
a. Binary Association

It associates objects of exactly two classes; it is denoted by a line, or an arrow between the associated objects.

Example



Association "works-for" associates objects of exactly two classes



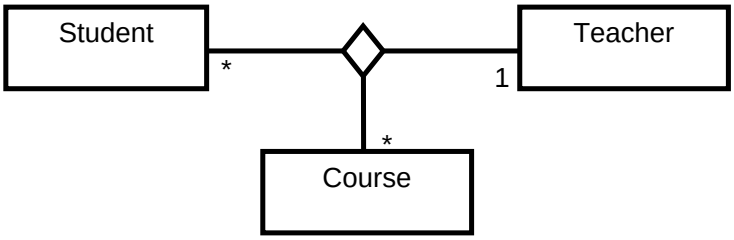
Association "drives" associates objects of exactly two classes

b. Ternary Association

It associates objects of exactly three classes; it is denoted by a diamond with lines connected to associated objects.

Example

Objects of exactly three classes are associated



c. N-ary Association

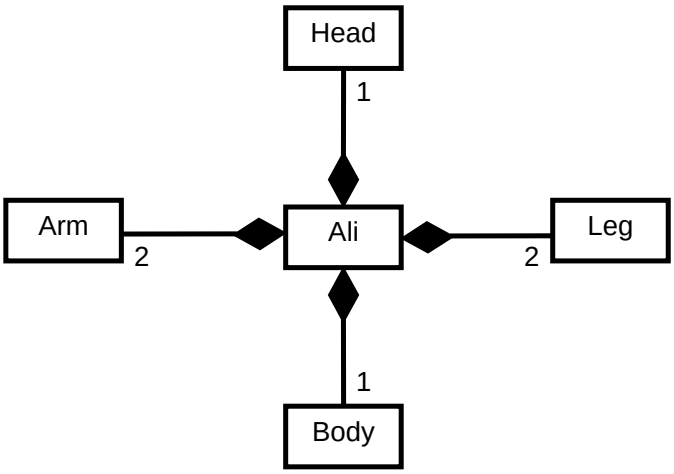
An association between 3 or more classes its practical examples are very rare.

05.4.Composition

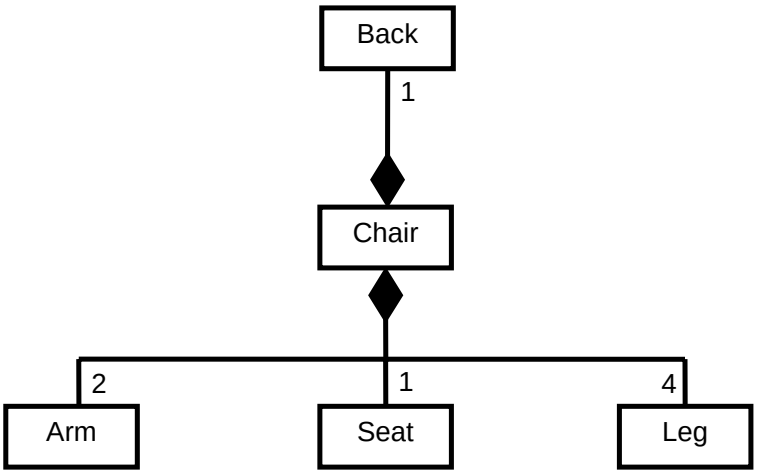
An object may be composed of other smaller objects, the relationship between the "part" objects and the "whole" object is known as Composition, Composition is represented by a line with a filled-diamond head towards the composer object

Example – Composition of Ali

© Virtual University of Pakistan



Example – Composition of Chair



Composition is stronger relationship:
Composition is a stronger relationship, because
Composed object becomes a part of the composer
Composed object can't exist independently

Example I

Ali is made up of different body parts

They can't exist independent of Ali

Example II

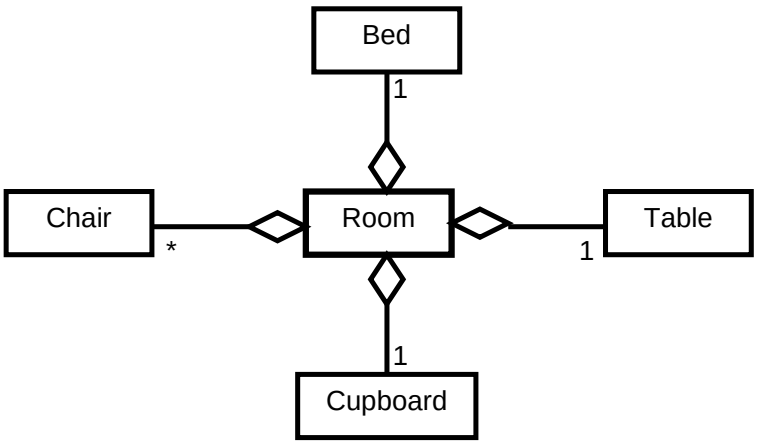
Chair's body is made up of different parts

They can't exist independently

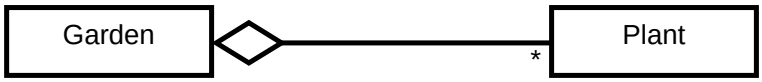
05.5. Aggregation

An object may contain a collection (aggregate) of other objects, the relationship between the container and the contained object is called aggregation, Aggregation is represented by a line with unfilled-diamond head towards the container

Example – Aggregation



Example – Aggregation



Aggregation is weaker relationship

Aggregation is weaker relationship, because

- Aggregate object is not a part of the container
- Aggregate object can exist independently

Example I
Furniture is not an intrinsic part of room
Furniture can be shifted to another room, and so can exist independent of a particular room

Example II
A plant is not an intrinsic part of a garden
It can be planted in some other garden, and so can exist independent of a particular garden

http://www.codeproject.com/KB/cpp/oopuml.aspx

© Virtual University of Pakistan

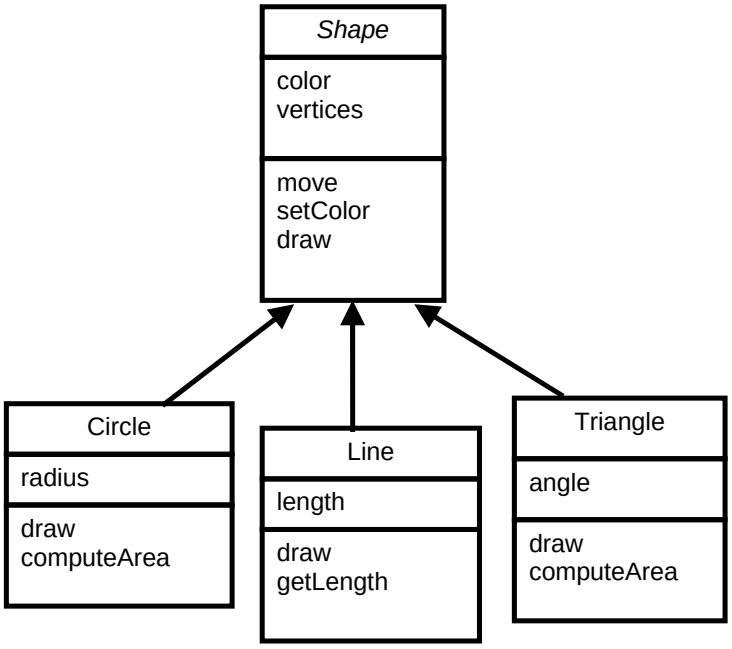
Lecture No.06

06.1.Class Compatibility

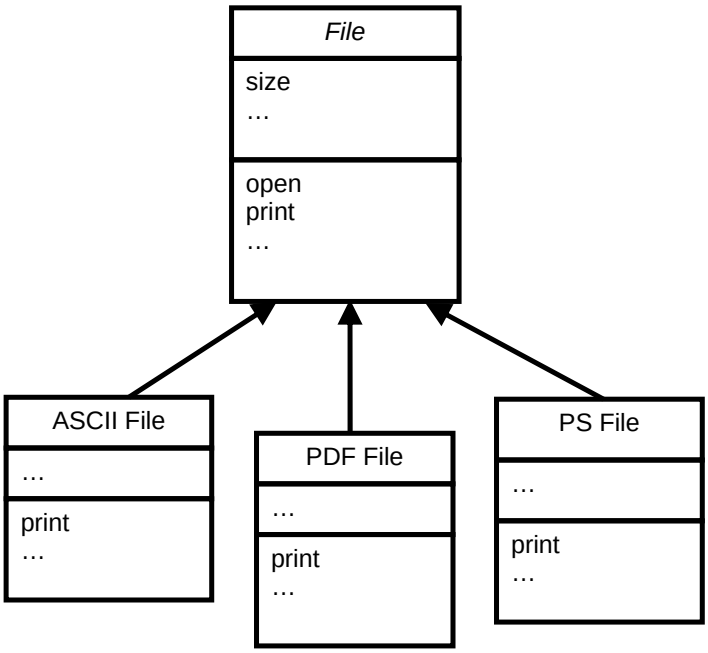
A class is behaviorally compatible with another if it supports all the operations of the other class. Such a class is called subtype. A class can be replaced by its subtype. Derived class is usually a subtype of the base class. It can handle all the legal messages (operations) of the base class. Therefore, base class can always be replaced by the derived class.

Examples

Child class also includes characteristics of its base class.



All the three derived class are behaviourally compatible with base class.



Wherever the file class is it can be replaced by any of its child classes.

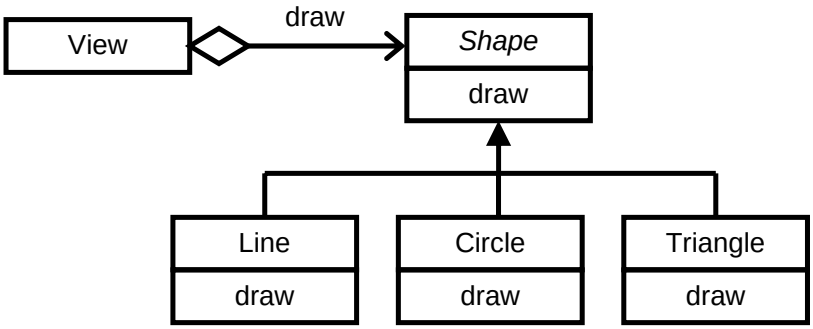
06.2.Polymorphism

It is also essential component of object oriented modeling (paradigm).
In general, polymorphism refers to existence of ***different forms*** of a single entity. For example, both Diamond and Coal are different forms of Carbon.

06.3.Polymorphism in OO Model

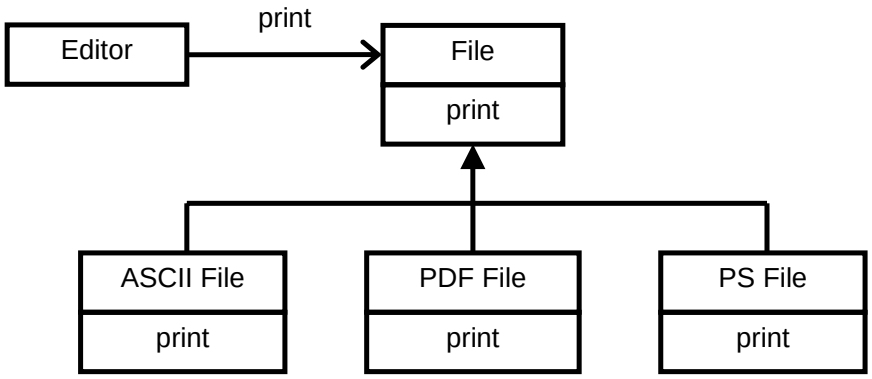
In OO model, polymorphism means that different objects can behave in different ways for the same message (stimulus). Consequently, sender of a message does not need to know exact class of the receiver.
Sender sends message to receiver and appropriate method is called on receiver side.

Example – Polymorphism



© Virtual University of Pakistan

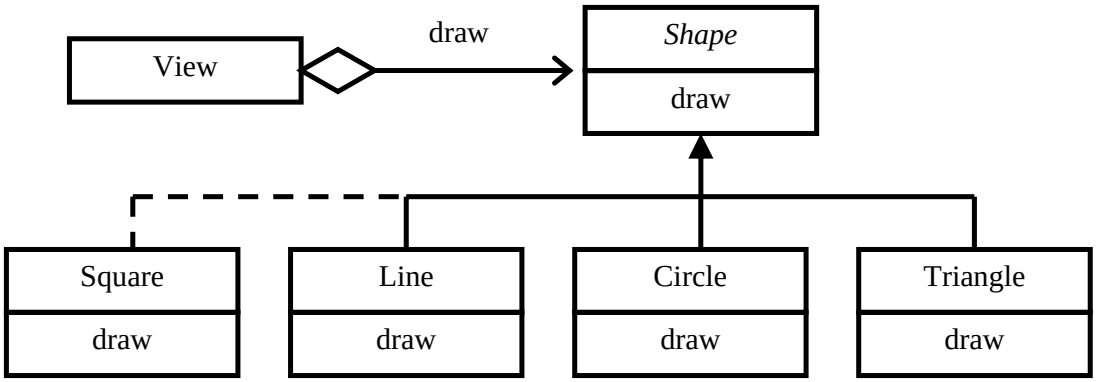
Shape class hierarchy shape is base class and there are three child classes line circle , triangle. View send draw method to shape class and draw is called according to the nature of actual object present.



Editor sends message print to file class and print is called based on the actual child object of file class message is same and appropriate execution will be done.

06.4.Polymorphism – Advantages

Messages can be interpreted in different ways depending upon the receiver class
New classes can be added without changing the existing model



In general, polymorphism is a powerful tool to develop flexible and reusable systems

06.5.Object-Oriented Modeling an Example

Problem Statement

Develop a graphic editor that can draw different geometric shapes such as line, circle and triangle. User can select, move or rotate a shape. To do so, editor provides user with a menu listing different commands. Individual shapes can be grouped together and can behave as a single shape.

Identify Classes

Extract nouns in the problem statement

Develop a graphic **editor** that can draw different geometric **shapes** such as **line**, **circle** and **triangle**. **User** can select, move or rotate a **shape**. To do so, **editor** provides **user** with a **menu** listing different **commands**. Individual **shapes** can be grouped together and can behave as a single **shape**.

Eliminate irrelevant classes

Editor – Very broad scope. But it is the name of overall system and we are going to model it so we will not make its object. For example if we are going to model computer we will not make its object but its components however if it is component of some other system then it will behave as an object. So it is marked as irrelevant.

User – Out of system boundary, it is interacting with the system from outside of the system.

Add classes by analyzing requirements

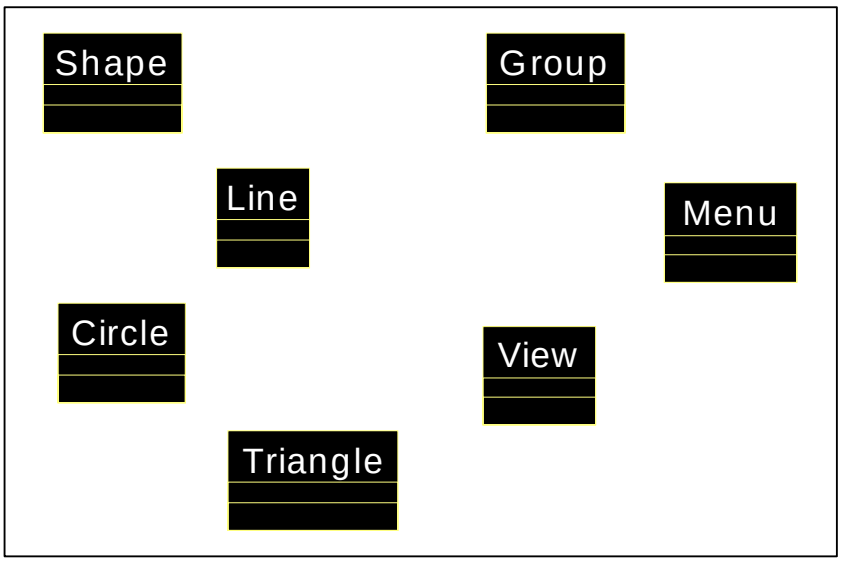
Group (of shapes) – required to behave as a shape so it should behave as an object in our system

"Individual shapes can be grouped together and can behave as a single shape"

View – graphic editor must have a display area to show the shapes. We made this object using domain knowledge.

- Shape
- Line
- Circle
- Triangle
- Menu
- Group
- View

So we have the following classes,



© Virtual University of Pakistan

Finding Associations:

Next step is to find associations between the objects.

Identify Associations

Find relationships between objects,

1. **Extract verbs connecting objects,**

*"Individual shapes can be **grouped together**"*

- Group **consists of** lines, circles, triangles
- Group can also **consists of** other groups (Composition)

Line, circle and triangle have composition relationship.

2. **Verify access paths**

   a. *View contains (draws) shapes*

      - *View contains lines*
      - *View contains circles*
      - *View contains triangles*
      - *View contains groups*

So there is **Aggregation relationship** between shapes and View.

Menu sends message to View

So there is Simple **One-Way Association relationship** between Menu and View.

Identify Attributes of the identified objects

Extract properties of the object,

a. From the problem statement
   Properties are not mentioned
b. From the domain knowledge

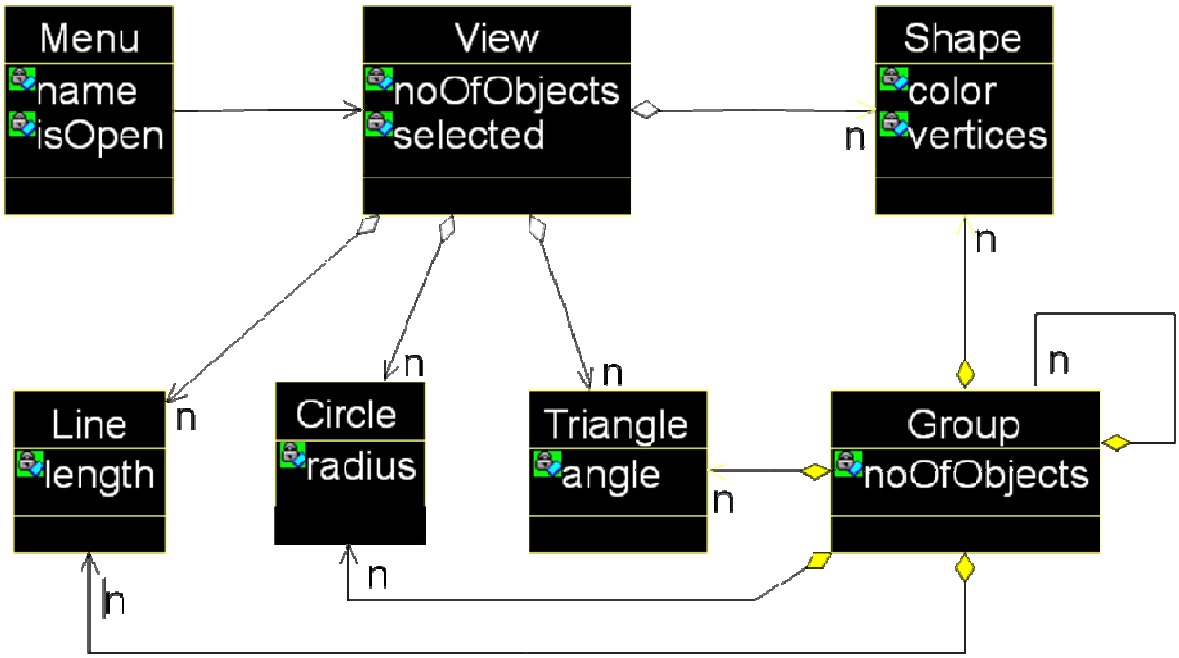
   - Line
     i. Color
     ii. Vertices
     iii. Length
   - Circle
     i. Color
     ii. Vertices
     iii. Radius
   - Triangle

i. Color
ii. Vertices
iii. Angle

- Shape
  i. Color
  ii. Vertices
- Group
  i. noOfObjects
- View
  i. noOfObjects
  ii. selected
- Menu
  i. Name
  ii. isOpen

Object Model – Graphic Editor

Object model so far is shown below,



Identify Operations

Extract verbs connected with an object

Develop a graphic editor that can **draw** different geometric shapes such as line, circle and triangle. User can **select**, **move** or **rotate** a shape. To do so, editor **provides** user with a menu listing different commands. Individual shapes can be **grouped** together and can **behave** as a single shape.

Eliminate irrelevant operations

 © Virtual University of Pakistan

Develop – out of system boundary
Behave – have broad semantics

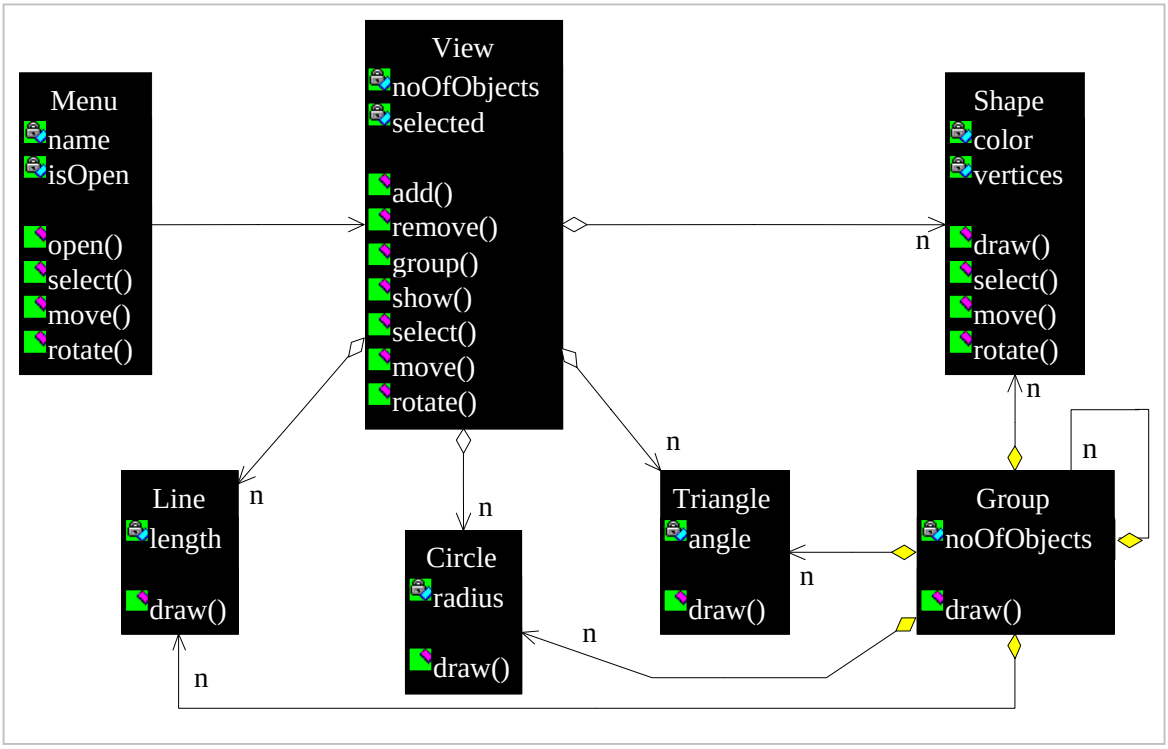
Following are selected operations:

- Line
  - Draw
  - Select
  - Move
  - Rotate
- Circle
  - Draw
  - Select
  - Move
  - Rotate
- Triangle
  - Draw
  - Select
  - Move
  - Rotate
- Shape
  - Draw
  - Select
  - Move
  - Rotate
- Group
  - Draw
  - Select
  - Move
  - Rotate
- Menu
  - Open
  - Select
  - Move
  - Rotate

Extract operations using domain knowledge

- View
  - Add
  - Remove
  - Group
  - Show
  - Select
  - Move
  - Rotate

Object Model after adding operations:



Identify Inheritance

a. Search lines like "is a kind of" by looking at keywords like "such as", "for example", etc

"…shapes such as line, circle and triangle…"

⇨ Line, Circle and Triangle inherits from Shape

By analyzing requirements

"Individual shapes can be grouped together and can behave as a single shape"

⇨ Group inherits from Shape

Refining the Object Model

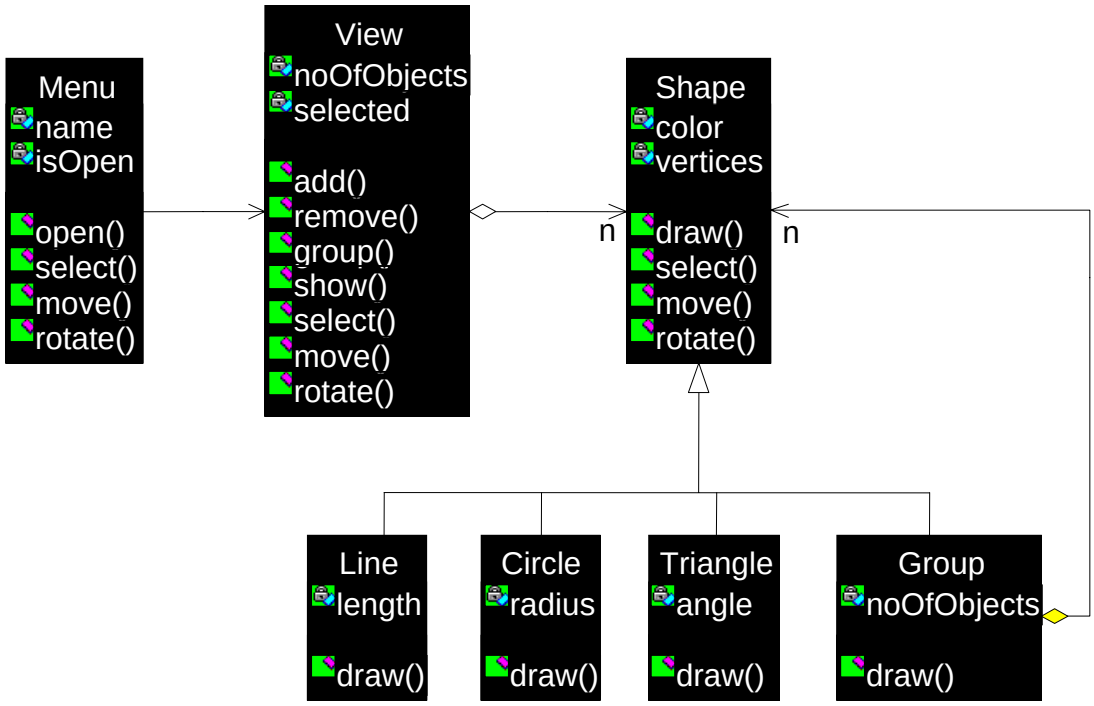
Application of inheritance demands an iteration over the whole object model
In the inheritance hierarchy,

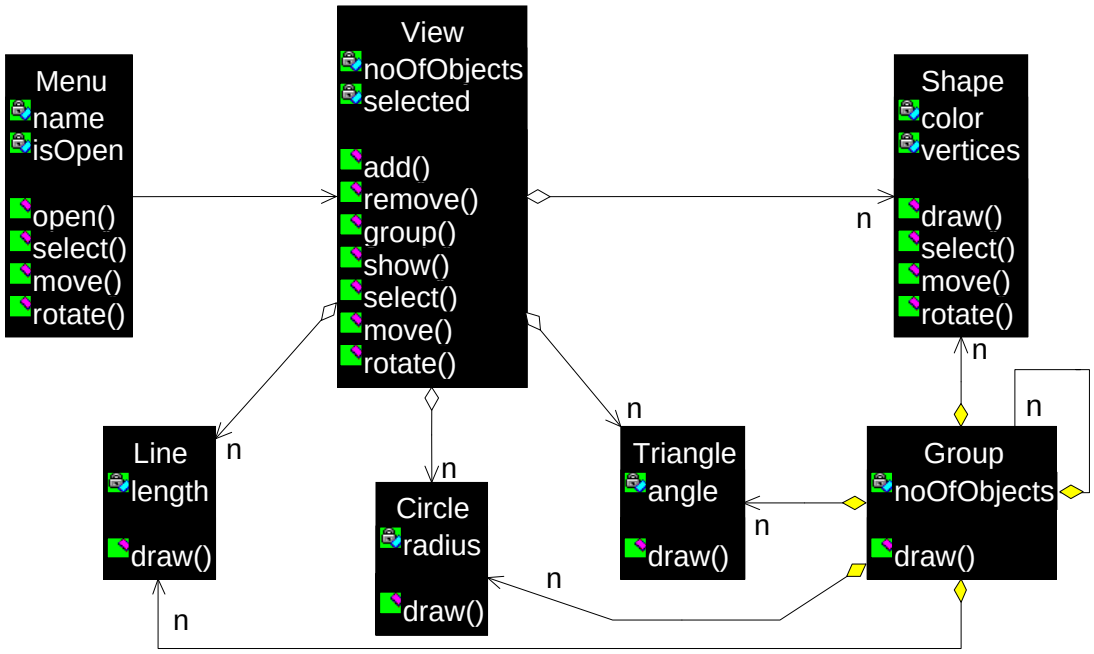
- All attributes are shared
- All associations are shared
- Some operations are shared
- Others are overridden

© Virtual University of Pakistan

Refining the Object Model

- Share associations
  - View contains all kind of shapes
  - Group consists of all kind of shapes
- Share attributes
  - Shape – Line, Circle, Triangle and Group
    - Color, vertices
- Share operations
  - Shape – Line, Circle, Triangle and Group
    - Select
    - Move
    - Rotate
- Share the interface and override implementation
  - Shape – Line, Circle, Triangle and Group
    - Draw





 © Virtual University of Pakistan

Lecture No.07

The basic concept "**Object**" of *Object Orientation (thinking in terms of objects)* is realized using classes in programming languages.

07.1.Class

It is a way (Mechanism) given by c++ to realize objects in a program. It is concrete implementation of objects in c++. We capture any object attributes and behaviour in a programming language using classes.
In other words it can be defined as facility given by c++ to create new types according to our requirement. (Class is composite data type made from basic c++ types like integers, chars and float).

Example:
Consider the examples of entity lion there are many lions but all lions will have similar attributes and behaviour.
Similarly consider student object all students have separate existence but all students have similar attributes and they exhibit similar behaviour.

When we hear word student or think about student a sketch comes in our mind for student along with its attributes and behaviour. The attributes of student comes in our mind are its name, roll no, class, degree so on. Similarly the behaviour of student comes in our mind are study, register and many more.

We need to capture the characteristic features of any object (attributes and behaviour) in the programming language. The concept of class is used for this purpose.

Now consider the scenario having many interacting objects: a **University System** having many objects like student, subject, classroom, and teacher so on…we will realize all these objects in our software using classes. These all object will use the services of each other for example student will ask teacher to teach him. This approach is closer to real life instead of having simple functions being called from main here these objects will call each other to get their services. This is the reason *we say that object oriented programming mimics real life.*

Uses

Objects are structured in terms of class so our problem becomes easier to understand in the terms c++ program.
We can implement interactions easily in terms of classes.

Student objects will interact with each other to take and give services to each other as happens in real life and mapped in object oriented programming approach.

Now we see how class mechanism helps us in implementing real life concept.

07.2. Type in C++

We implement generic concepts using types.

We have to model generic concept of Student. But there is no built in type **student** in c++ like built-in c++ type's **int** or **float**. Class is mechanism in c++ that will allow us to define student as user defined type, similarly generic concept circle will also be implemented in the same way. User define types will be,

- Student in student management system
- Circle in a drawing software

As objects have attributes and behaviour so corresponding classes will also have data members and methods as shown below,

Ali	Corresponding class
Characteristics (attributes)	/*c++ code for class Person, we can create any object like Ali from it*/
Name Age	class Person { private: /* attributes are generally made private*/ char name[]; /*char array to store name*/ int age; /*int age to store age*/
Behavior (operations)	public: /* methods are generally made public*/
Walks Eats	Person(); /*constructor used to initialize data members*/ void walks(); /* method walk */ void eats(); /*method eats*/ }
a. object	**b. class code**

07.3. Abstraction

We only include those details in the system that are required for making a functional system so we will leave out irrelevant attributes and behaviour from our objects.
Take the example of student,

Student

- Name
- Address
- Sibling
- Father Business

} **Relevant to our problem**

} **Not relevant to our problem**

07.4. Defining a New User Defined Type

There are two ways to create user defined types for objects in c++ these are,

Structure Definition:

Partially we can use Structures to define an object

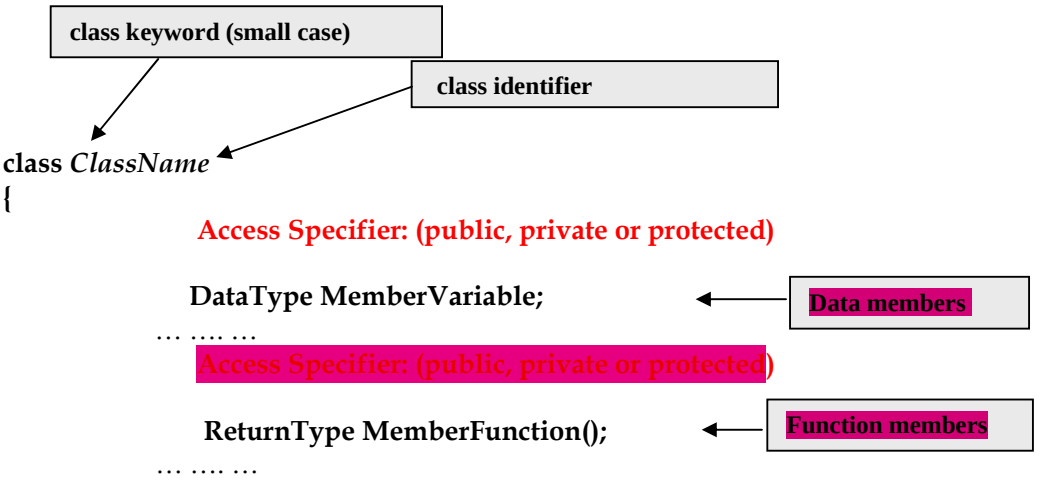
```
Struct{

};
```


© Virtual University of Pakistan

In c we can not define functions in a structure however in c++ we can add functions in both structure and classes.

Class Definition:



Example

```
class Student
{
    private:
        int rollNo;
        char *name;
        float CGPA;
        char *address;

    public:
        void setName(char *newName);
        void setRollNo(int newRollNo);

};
```

Member Variables

Member Functions

Why Member Functions:
They model the behaviors of an object,

Objects can make their data invisible (in accordance with the principle of data hiding). Setters and getters functions are provided by class to access the its members it also minimizes the changes to move the objects in inconsistent state as we can write checks in our setter functions for example we can check that whether the user has entered correct age value and has not entered negative value for age.
Object remains in consistent state

Example:

We can check that the entered roll number by user is positive or negative,

```
Student aStudent;
aStudent.rollNo = 514;
aStudent.rollNo = -514;          //Error
```

07.5. Object and Class:

Object is an instantiation of a user defined type or class. Once we have defined a class we can create as many objects for that class as we require.

Declaring class variables

Variables of classes (objects) are declared just like variables of structures and built-in data types as follows,

TypeName VariableName;

int var; // declaring built in int data type variable
Student aStudent; // declaring user defined class Student object

07.6. Accessing members

Members of an object can be accessed using,

a. dot operator (.) to access via the variable name

```
Student aStudent;   // declaring Student object
aStudent. rollNo = 5;
```

b. arrow operator (->) to access via a pointer to an object

```
Student * aStudent = new Student();
// declaring and initializing Student pointer
aStudent->rollNo = 5;
```

Note: it is against the principle of OOP to access the data members directly using object of class as we have done above. This code is given for example only we should write assessor functions (setters and getters) wherever we want to access the members of the class.

Member functions are accessed in the similar way using dot or arrow operator.

Example

```
class Student{
  int rollNo;
  void setRollNo(int aNo);
};

  Student aStudent;
  aStudent.setRollNo(5);
```


© Virtual University of Pakistan

```
Student *ptr_student = new Student();
ptr_student->setRollNo(5);
```

07.7. Access specifiers

These are used to enforce access restrictions to members of a class, there are three access specifiers,

1. **'public'** is used to tell that member can be accessed whenever you have access to the object
2. **'private'** is used to tell that member can only be accessed from a member function
3. **'protected'** to be discussed when we cover inheritance

Example

```
class Student{
private:
        char * name;
        int rollNo;
public:
        void setName(char *);
        void setRollNo(int);
...
};
```

Can not be accessed outside the class

Can be accessed outside the class

Example Program

```
class Student{
char * name;
        int rollNo;
public:
        void setName(char *);
void setRollNo(int aNo);
};
void Student::setName(char * aName){
if (strlen(aName) > 0)
    {
        name = new char[strlen(aName)];
        strcpy(name,aName);
    }
}

void Student::setRollNo(int arollNo){
if(arollNo > 0)
rollNo = arollNo;
}

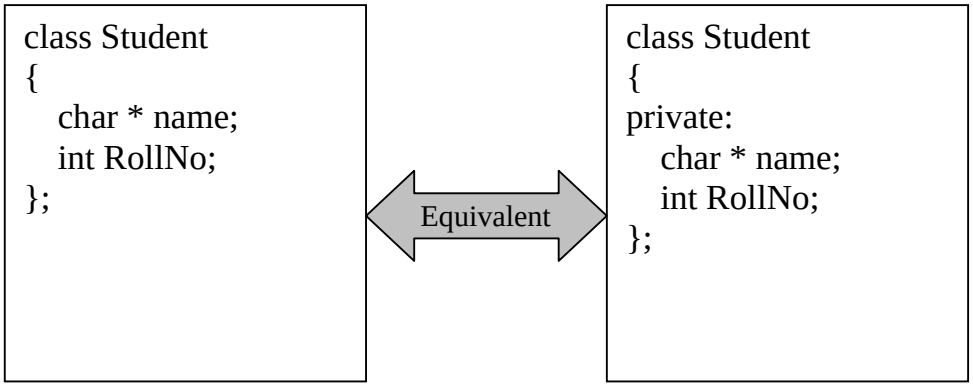
int main(){
        Student aStudent;
 aStudent.rollNo = 5;
```

```
/* Error: we can not access private member of the class. */
 aStudent.name  = "Ali";
/* Error: we can not access private member of the class */
        aStudent.setRollNo(1);
        aStudent.setName("Ali");
/* Correct way to access the data member using public setter functions */
}
```

Default access specifier

When no access specifier is mentioned then default access specifier is **private**.

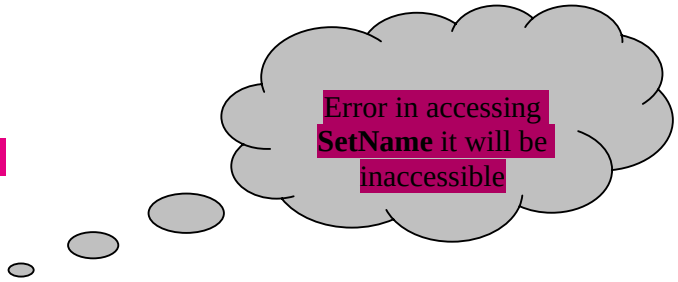
Example



Example

We should use keyword public before the methods of the class as given below of will not use public keyword they will also be treated as private ad will not be accessible outside the class as shown below,

```
class Student
{
        char * name;
        int RollNo;
        void SetName(char *);
};
Student aStudent;
aStudent.SetName(Ali);
```



Corrected code will be,

```
class Student
{
        char * name;
        int RollNo;
public:
        void setName(char *);
```


© Virtual University of Pakistan

```
};
Student aStudent;
aStudent.SetName("Ali");
```

Lecture No.08

08.1.Member Functions

- Member functions are the functions that operate on the data encapsulated in the class
- Public member functions are the interface to the class

08.2.Defining Member Functions

We can define member functions in two ways,

a. We can define member functions of the class inside the class definition when we define any class in our program.

OR

b. We declare member function inside the class definition and declare them outside the class.

In this case class definition is added in ClassName.h file and class implementation code is added in ClassName.cpp file.

Function definition inside the class:

General Syntax:

```
class ClassName {
        …
        public:
        ReturnType FunctionName() {
                …
        }
};
```

Example:

Define a class of student that has a roll number. This class should have a function that can be used to set the roll number

```
class Student{
        int rollNo;
public:
        void setRollNo(int aRollNo){
                rollNo = aRollNo;
        }
};
```

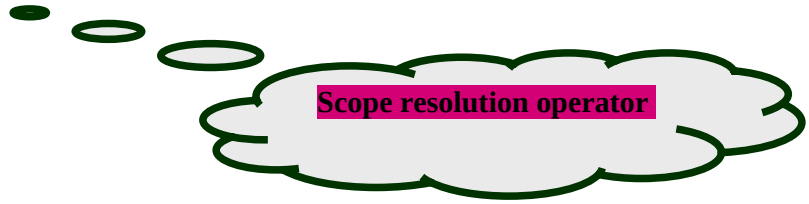
Function definition outside class

General Syntax:

```
class ClassName {
        …
```


© Virtual University of Pakistan

```
        public:
        ReturnType FunctionName();
};
ReturnType ClassName::FunctionName()
{
        …
}
```



Example

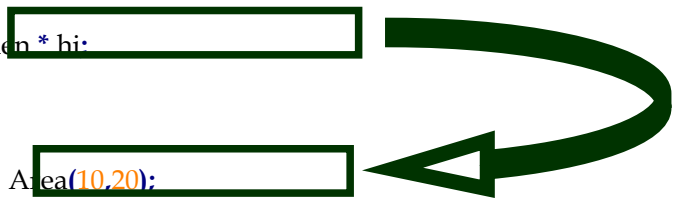
```
class Student{
        …
        int rollNo;
public:
        void setRollNo(int aRollNo);
};
void Student::setRollNo(int aRollNo){
        …
        rollNo = aRollNo;
}
```

08.3.Inline Functions

- Inline functions is a way used by compilers to improve efficiency of the program, when functions are declared inline normal process of function calling (using stack) is not followed instead function code is added by compiler at all points where these functions have been called. Basic concept behind inline functions is that they are functions in our code but in compiler generated files these functions code is added by compiler at all places where they were called in the code.
- Normally small size functions that need to be called many times during program execution are declared inline. Inline functions decrease the code execution time because program in their case doesn't involve function call overhead.
- Keyword 'inline' is used to request compiler to make a function inline.
- However using inline keyword with function doesn't guarantee that function will definitely in inlined, it depends on the compiler if it finds it can make function inline it does so otherwise it ignores the keyword inline and treat the function as normal function.

Example

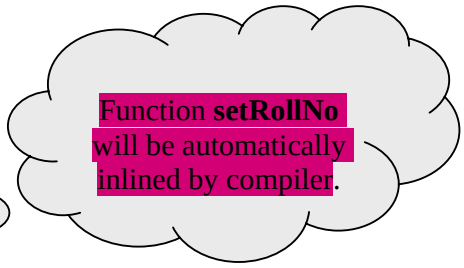
```
inline int Area(int len, int hi)
{
        return len * hi;
}
int main()
{
        cout << Area(10,20);
```



```
    return 0;
}
```

Inline Functions

The functions defined inside the class are by default inline (whether we mention keyword inline with them or not)
In case we define function outside the class then we must use the keyword 'inline' to make the function inline.
However compiler decides whether it will implement these functions code as inline or not.



Example
Inline function inside the class:

```
class Student{
        int rollNo;
public:
        void setRollNo(int aRollNo){
                …
                rollNo = aRollNo;
        }
};
```

Example
Inline function outside the class:

```
class Student{
        …
        public:
        inline void setRollNo(int aRollNo);
};
void Student::setRollNo(int aRollNo){
        …
        rollNo = aRollNo;
}
class Student{
        …
        public:
        void setRollNo(int aRollNo);
};
inline void Student::setRollNo(int     aRollNo){
        …
        rollNo = aRollNo;
}
class Student{
        …
        public:
```


© Virtual University of Pakistan

```
        inline void setRollNo(int aRollNo);
};
inline void Student::setRollNo(int
        aRollNo){
        …
        rollNo = aRollNo;
}
```

08.4.Constructor

Constructor is used to **initialize** the objects of a class. Constructor is used to ensure that object is in well defined state at the time of creation.
The constructor of a class is automatically generated by compiler however we can write it by our self also.
Constructor is automatically called when the object is created. Constructors are not usually called explicitly by us.

08.5.Constructor Properties

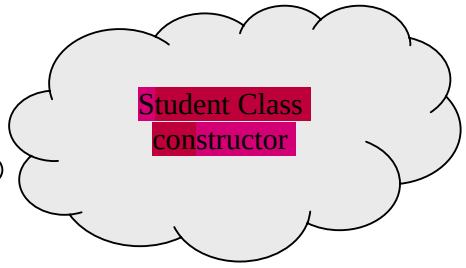
- Constructor is a special function having same name as the class name
- Constructor does not have return type
- Constructors are commonly public members

Example

```
class Student{
        int rollNo;
public:
        Student(){
                rollNo = 0;

        }
};

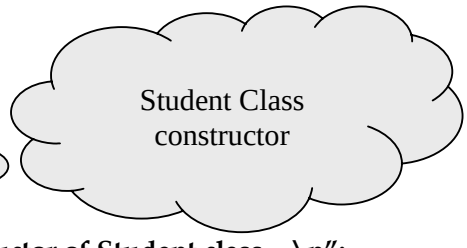
int main()
{
        Student aStudent;
        /*constructor is implicitly called at this point*/
}
```



We can assure that constructor is called automatically by adding cout statement in constructor as shown below,

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

class Student{
        int rollNo;
public:
        Student(){
                rollNo = 0;
                cout<<"I am constructor of Student class…\n";
```



```
        }
};

int main()
{
        Student aStudent;
        /*constructor is implicitly called at this point*/
system("pause");
return 0;
}
```

08.6. Default Constructor

- Constructor without any parameter or with all parameters with default values is called default constructor
- If we do not define a default constructor the compiler will generate a default constructor
- Compiler generated default constructor is called implicit and user written default constructor is called explicit
- This compiler generated default constructor initialize the data members to their default values
- If we have given any constructor for a class whether it is
    - our own **explcit default constructor** (i.e parameterless or with parameters having default values)
      or
    - our own constructor with parameters

Then compiler will not create implicit default constructor[5].

Example

Consider the class student below it has no constructor so compiler will generate one for it,

```
class Student
{
        int rollNo;
        char *name;
        float GPA;
public:
        …               //no constructors
};
```

Code of Compiler generated implicit default constructor

```
{
```

[5] compiler generated default constructor is called implicit and user written default constructor is called explicit

© Virtual University of Pakistan

```
        rollNo = 0;
        GPA = 0.0;
        name = NULL;
}
```

08.7.Constructor Overloading

We can write constructors with parameters as well. These parameters are used to initialize the data members with user supplied data (passed as parameter). The example is shown below, here example Student class has four constructors their prototypes are,

1. **Student();** /* explicit default parameterless constructor */
2. **Student(char * aName);** /* constructor with one parameter* /
3. **Student(char * aName, int aRollNo);** /* constructor with two parameters */
4. **Student(int aRollNo, int aRollNo, float aGPA);** /* constructor with three parameters */

Example

```
class Student{
        int rollNo;
        char *name;
        float GPA;
public:
        Student();  /* explicit default constructor */
        Student(char * aName);  /* constructor with one parameter* /
        Student(char * aName, int aRollNo);  /* constructor with two parameters */
        Student(int aRollNo, int aRollNo, float aGPA);  /* constructor with three parameters */

};
Student::Student(){

        rollNo = 0;
        name = NULL; // to indicate it is pointing to nothing at this moment otherwise it can generate erroneous code.
        GPA = 0.0;

}
Student::Student(int aRollNo){

        if(aRollNo < 0){
                rollNo = 0;
        }
        else {
        rollNo = aRollNo;
        }
name = NULL;
```

```
}
Student::Student(int aRollNo,
                                    char * aName){

        if(aRollNo < 0){
                rollNo = 0;
        }
        else {
        rollNo = aRollNo;
        }

 if (strlen(aName) > 0)
                                {
        name = new char[strlen(aName)+1];
           strcpy(name,aName);
                                }
 else
       {
name = NULL;
}

}
```

We can create this Student class object using any one of these constructors as follows,

```
int main()
{
        Student student1;  // default constructor will be used
        Student student2("Name"); // one parameter constructor will be used
        Student student3("Name", 1);  // two parameter constructor will be used
        Student student4("Name",1,4.0); // three parameter constructor will be used
}
```

08.8. Constructor Overloading

We can use default parameter values to reduce the writing effort in that case we will have to write only one constructor and it will serve the purpose of all constructors as given below,

Example

```
Student::Student(       char * aName = NULL,  int aRollNo= 0,  float aGPA = 0.0) {
…

}
```

It is equivalent to all three constructors,

© Virtual University of Pakistan

Student();
Student(char * aName);
Student(char * aName, int aRollNo);
Student(char * Name, int aRollNo, float aGPA);

It will use default values if values are not passed as arguments while creating objects) it is described in code given below,

```
int main()
{
    Student student1;  /*char * aName = NULL,  int aRollNo= 0, float aGPA = 0.0*/

    Student student2( "Name" );  /*char * aName = Name, int aRollNo= 0, float aGPA = 0.0*/

    Student student3( "Name",  1 );  /*char * aName = Name, int aRollNo= 1, float aGPA = 0.0*/

    Student student4( "Name", 1 , 4.0);  /*char * aName = Name, int aRollNo= 1, float aGPA = 4.0*/


}
```

08.9.Copy Constructor
Copy constructors are used when:

- Initializing an object at the time of creation (we want to create an object with state of a pre existing object)
- When an object is passed by value to a function (As you know temporary copy of object is created on stack so we need copy constructor to create that temporary object with the state of actual object being passed).

Example

```
void func1(Student student){
…
}
```

```
int main(){

        Student studentA;
        Student studentB = studentA;
```

Copy constructor will be called to create temporary student object

```
        func1(studentA);

}
```

Copy constructor will be called as we are creating studentB in terms of studentA.

Copy Constructor (Syntax)

```
Student::Student( const Student &obj){

    /*copying values to newly created object*/
   rollNo = obj.rollNo;
        name = obj.name;
        GPA = obj.GPA;

}
```

As was the case with default constructor compiler also generates copy constructor by itself however we can override that copy constructor by writing our own copy constructor as shown in example below,

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

class Student{
        int rollNo;
public:
        Student(){
                rollNo = 0;
                cout<<"I am default construct…\n";
        }

Student(const Student &obj){
         cout<<"I am copy constructor of Student class\n";
  rollNo = obj.rollNo;

}
};

int main()
{
        Student aStudent;
        /*default constructor is implicitly called at this point*/

        Student bStudent = aStudent;
        /*copy constructor is implicitly called at this point*/

system("pause");
```

Student class default constructor

Student class copy constructor

© Virtual University of Pakistan

```
return 0;
}
```

08.10. Shallow Copy

- When we initialize one object with another then the compiler copies state of one object to the other using copy constructor by assigning data member values of previous object to newly created object.

Shallow copy using default Copy Constructor (Syntax)

```
Student::Student(     const Student & obj ){

        rollNo = obj.rollNo;
        name = obj.name;
        GPA = obj.GPA;
}
```

- This kind of copying is called shallow copying

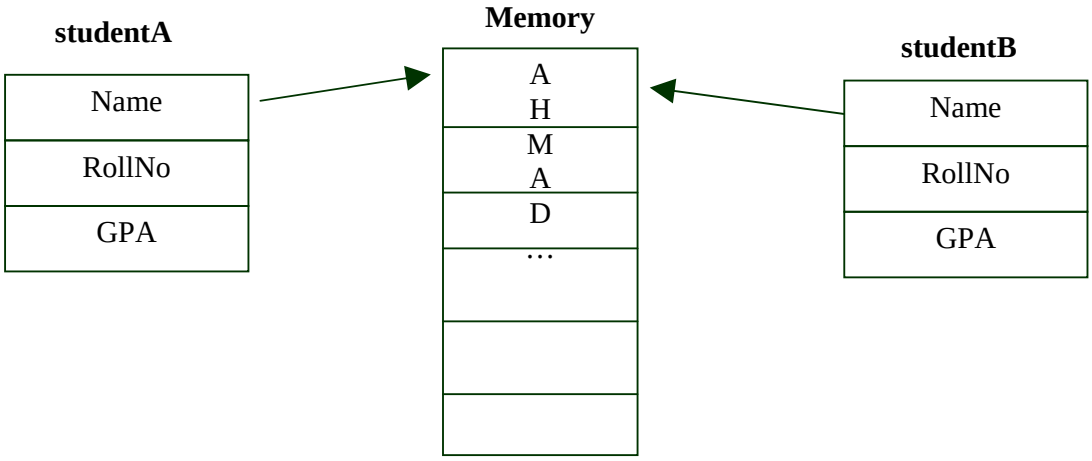
Example

```
Student studentA;
Student studentB = studentA;
```
/*Shallow copy: compiler will use copy constructor to assign studentA values to newly created object studentB*/

Shallow copy works fine if our class doesn't include dynamic memory allocation but in case of dynamic memory allocation it leads to dangling pointer problem as explained below,

Problem is Shallow Copy

Student class data member name of char * type is added to store the name of student and it is using **dynamic memory** according to the length of name entered by user for student.



Object studentB is also pointing to memory allocated object studentA for Student class data member **name** of char * type, now there are two problems with this sort of copying,

Suppose we delete first object studentA for some reason then its destructor will also free memory allocated by it hence memory area containing name "AHMAD" will also be freed and will be given to some other application by operating system, but studentB member **name** is still pointing to that area so issue of *"Dangling Pointer"* will arose. [Pointer pointing to incorrect memory location]. If we will try to print the name of object studentB our program will terminate abnormally as it was pointing memory of some other applications.
Secondly if for some reason we change name of studentA the value of object studentB will also be changed as it pointing to same memory location.

We resolve these two issues using deep copy.

08.11. Deep Copy

We write our own deep copy code in copy constructor so that when we create new object from an existing object using copy constructor we also allocate new dynamic memory for data members involving dynamic memory as shown below,

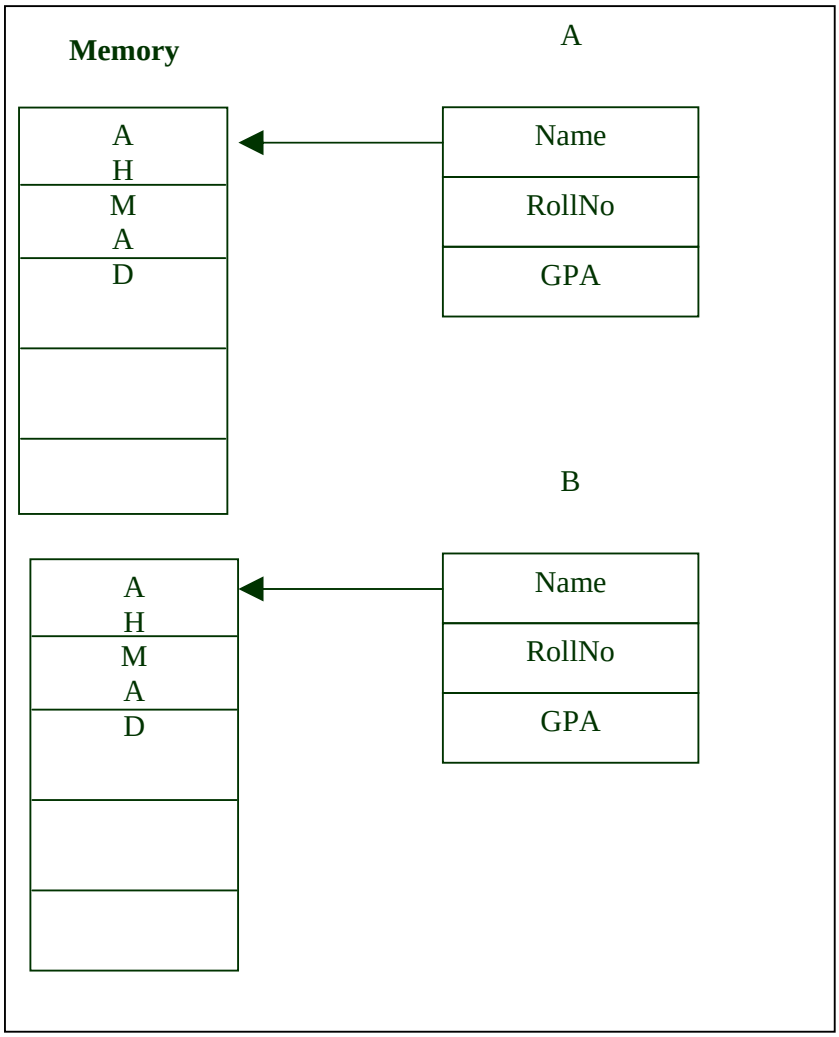
```
Student::Student( const Student & obj){

        int len = strlen(obj.name);
        name = new char[len+1];
        // assignming new dynamic memory to data member name of char * type for
newly created object*/
        strcpy(name, obj.name);
        …
        //copy rest of the data members in the same way
}
```


© Virtual University of Pakistan

Example

```
Student studentA;
Student studentB = studentA; // now copy constructor will perform deep copy (separate memory for both objects)
```



In case our class doesn't involve dynamic memory then default copy constructor that performs shallow copy works fine.
In case our class has any data member involving dynamic memory we have to write our own code to do deep copy.

Lecture No.09

Review

Copy Constructor

Copy constructors are used when:

Initializing an object at the time of creation (we want to create an object with state of a pre existing object)

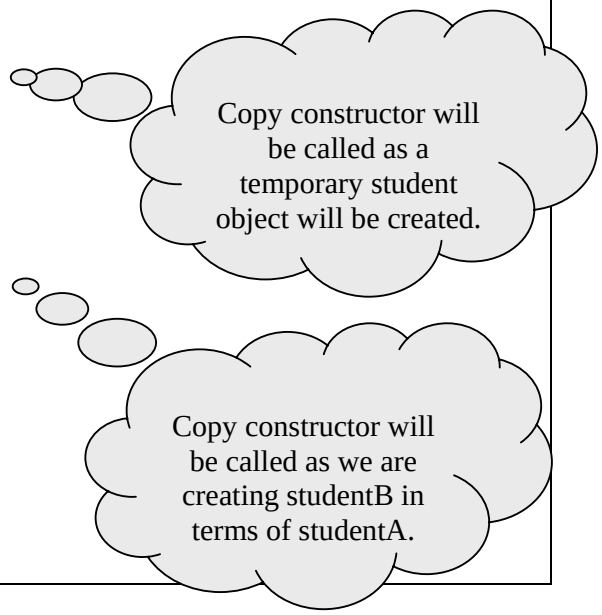
When an object is passed by value to a function (As you know temporary copy of object is created on stack so we need copy constructor to create that temporary object with the state of actual object being passed).

Example

```
void func1(Student student){
…
}

int main(){

        Student studentA;
        Student studentB = studentA;
        func1(studentA);
        return 0;
}
```



Copy Constructor (Syntax)

```
Student::Student( const Student &obj){

   /*copying values to newly created object*/
  rollNo = obj.rollNo;
        name = obj.name;
        GPA = obj.GPA;
}
```

As was the case with default constructor compiler also generates copy constructor by itself however we can override that copy constructor by writing our own copy constructor as shown in example below,

As was the case with default constructor compiler also generates copy constructor by itself however we can override that copy constructor by writing our own copy constructor as shown in example below,

```
#include <iostream>
```

 © Virtual University of Pakistan

```
using namespace std;

class Student{
        int rollNo;
public:
        Student(){
                rollNo = 0;
                cout<<"I am default construct...\n";
        }

Student(const Student &obj){
         cout<<"I am copy constructor of Student class\n";
  rollNo = obj.rollNo;

}
};

int main()
{
        Student aStudent;
        /*default constructor is implicitly called at this point*/

        Student bStudent = aStudent;
        /*copy constructor is implicitly called at this point*/

system("pause");
return 0;
}
```

Student class default constructor

Student class copy constructor

09.1.Shallow Copy

When we initialize one object with another then the compiler copies state of one object to the other using copy constructor by assigning data member values of previous object to newly created object. This kind of copying is called shallow copying.

Shallow copy using default Copy Constructor (Syntax)

```
Student::Student(      const Student & obj ){

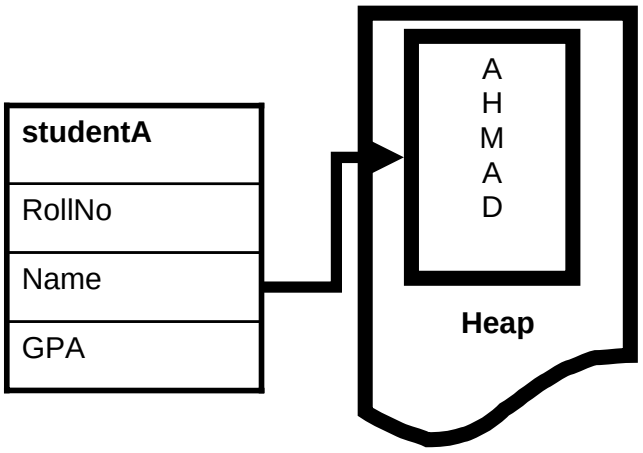
        rollNo = obj.rollNo;
        name = obj.name;
        GPA = obj.GPA;
}
```

This kind of copying is called shallow copying

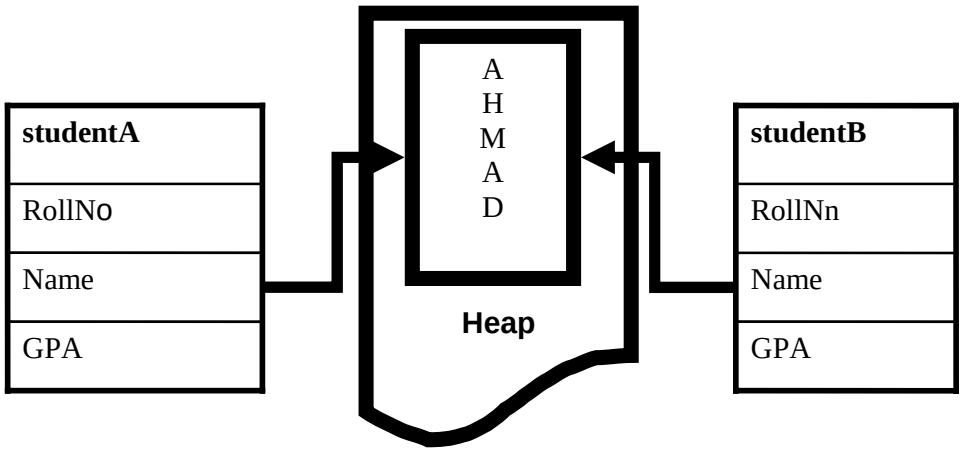
Example

Student studentA;
Student studentB = studentA; /*Shallow copy: compiler will use copy constructor to assign studentA values to newly created object studentB*/

Example
Student studentA("Ahmad");



Student studentB = studentA;




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

class Student{
    char * name;
        int rollNo;
public:
        Student(char * aName, int arollNo){
      name = new char[strlen(aName)+1];
      strcpy(name,aName);
```


© Virtual University of Pakistan

```
                rollNo = arollNo;

        }
  Student(const Student &obj){
    name = obj.name;
    rollNo = obj.rollNo;


  }
  void showName(){
     cout<<name<<endl;

     }
  ~Student(){

       delete []name;

       }
};

int main()
{
        Student studentA("AHMAD",1);
        Student studentB = studentA;
        /*copy constructor is implicitly called at this point*/

        studentA.showName();
  studentB.showName();

  system("pause");
  return 0;
}
```

Shallow copy works fine if our class doesn't include dynamic memory allocation but in case of dynamic memory allocation it leads to dangling pointer problem as explained below.

Problem is Shallow Copy

Student class data member name of char * type is added to store the name of student and it is using **dynamic memory** according to the length of name entered by user for student.

Student class data member **name** (*char* *) of object studentB is also pointing to memory allocated for datamember **name** of object studentA, due to this there may be two kinds of problems.

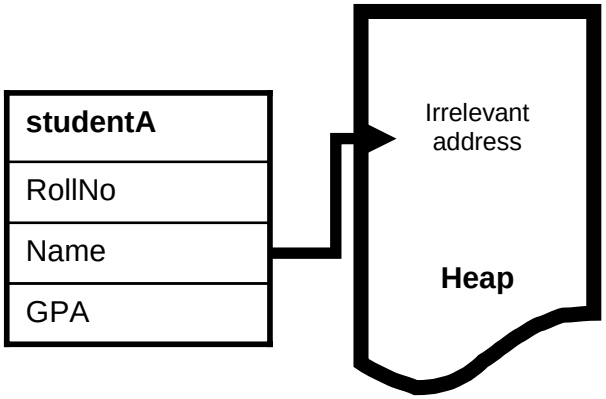
Suppose we delete first object studentA for some reason then its destructor will also free memory allocated by it hence memory area containing name "AHMAD" will also be freed and will be given to some other application by operating system, but

studentB member **name** is still pointing to that area so issue of *"Dangling Pointer"* [Pointer pointing to incorrect memory location] will arose.
Same will happen if object studentB is deleted then studentA object data member name will become dangling pointer. This has been explained below,
Let us change code in main to make our second object studentB in a new local scope as shown below,

```
int main(){
Student studentA("Ahmad",1);

{
   Student studentB = studentA;
}
return 0;
}
```

Now if we will try to print the name of object studentA our program will not show any output as name is pointing to some irrelevant memory address,



Complete program code is given below,

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

class Student{
    char * name;
        int rollNo;
public:
        Student(char * aName, int arollNo){
      name = new char[strlen(aName)+1];
      strcpy(name,aName);
                rollNo = arollNo;

        }
  Student(const Student &obj){
```


© Virtual University of Pakistan

```
    name = obj.name;
    rollNo = obj.rollNo;

  }
  void showName(){
     cout<<name<<endl;

     }
  ~Student(){
         delete []name;
         }
};

int main()
{
        Student studentA("AHMAD",1);

  {
       Student studentB = studentA;
       /*copy constructor is implicitly called at this point*/
  }

        studentA.showName();

  system("pause");
  return 0;
}
```

Secondly if for some reason we change name of object studentA the value of object studentB will also be changed as it pointing to same memory location.

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

class Student{
    char * name;
        int rollNo;
public:
        Student(char * aName, int arollNo){
     name = new char[strlen(aName)+1];
     strcpy(name,aName);
                rollNo = arollNo;

        }
  Student(const Student &obj){
    name = obj.name;
    rollNo = obj.rollNo;
```

```
   }

   void setName(char * aName){
      strcpy(name,aName);

      }

   void showName(){
      cout<<name<<endl;

      }
   ~Student(){

            delete []name;

            }
};

int main()
{
         Student studentA("AHMAD",1);
         Student studentB = studentA;
         /*copy constructor is implicitly called at this point*/

         studentA.showName();
         studentB.showName();


         studentA.setName("MOEEN");
         studentA.showName();
         studentB.showName();

   system("pause");
   return 0;
}
```

We resolve these two issues using deep copy.

09.2.Deep Copy

We write deep copy code in copy constructor so that when we create new object from an existing object using copy constructor we also allocate new dynamic memory for data members involving dynamic memory as shown below,

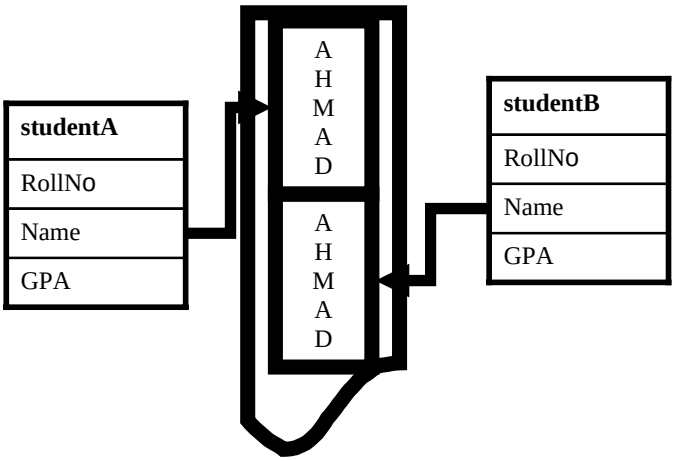
Student::Student(const Student & obj){

```
        int len = strlen(obj.name);
        name = new char[len+1]; // assignming new
/*dynamic memory to data member name of char * type for newly created object.*/
        strcpy(name, obj.name);
        …
        //copy rest of the data members in the same way
}
```

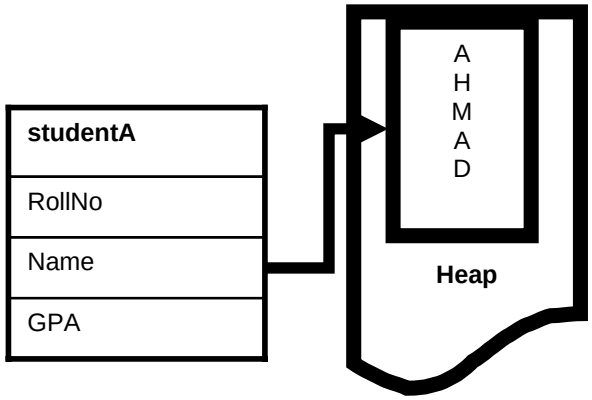
Now we see what happens when we created objects in main as shown below,

```
int main(){
Student studentA("Ahmad",1);

{
   Student studentB = studentA;
}
}
```



Now when we will execute code with object studentB in local scope our code still works fine and shows name for object studentA as now deletion of object studentB has no effect on object studentA as shown below,



Example

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

class Student{
   char * name;
         int rollNo;
public:
         Student(char * aName, int arollNo){
       name = new char[strlen(aName)+1];
       strcpy(name,aName);
                  rollNo = arollNo;

         }
   Student(const Student &obj){

      name = new char[strlen(obj.name)+1];
      strcpy(name,obj.name);
      rollNo = obj.rollNo;


   }

  void showName(){
       cout<<name<<endl;

       }
   ~Student(){

           delete []name;

           }
};

int main()
{
```

```
        Student studentA("AHMAD",1);
   {
       Student studentB = studentA;
       /*copy constructor is implicitly called at this point*/
   }
        studentA.showName();

   system("pause");
   return 0;
}
```

09.3. Important points about copy constructor:

1. In case our class doesn't involve dynamic memory then default copy constructor that performs shallow copy works fine.
2. In case our class has any data member involving dynamic memory we have to write our own code in copy constructor to perform deep copy.
3. Copy constructor is normally used to perform deep copy
4. If we do not make a copy constructor then the compiler performs shallow copy
5. Shallow copy performs bitwise copy.

09.4. Destructor

1. Destructor is used to free memory that is allocated through dynamic allocation. We have to free memory allocated using new operator by over self in destructor otherwise it remain occupied even after our program ends.
2. Destructor is used to perform house keeping operations.
3. Destructor is a function with the same name as that of class, but preceded with a tilde '~'

Example

```
class Student
{
        …
public:
        ~Student(){
                if(name){
                        delete []name;
                }
        }
};
```

Overloading

Destructors cannot be overloaded.

Sequence of Calls

Constructors and destructors are called automatically
Constructors are called in the sequence in which object is declared
Destructors are called in reverse order

Example

```
Student::Student(char * aName){
        …
                cout << aName << "Constructor\n";
        }
        Student::~Student(){
                cout << name << "Destructor\n";
        }
};
```

Example

© Virtual University of Pakistan

```
int main()
{
        Student studentB("Ali");
        Student studentA("Ahmad");
        return 0;
}
```

Example

Output:
Ali Constructor
Ahmad Constructor
Ahmad Destructor
Ali Destructor

09.5. Accessor Functions

In accordance to principle of information hiding data members of a class are declared as private so that outside world can not access the private data of the object only an interface is provided to outside world in the form of functions.
Accessor functions are also used to access private data of the object, we provide accessor functions to get and set private data members of the class.
We also add error checking code in accessor functions to reduce errors so that object doesn't move in illegal state.

Example – Accessing Data Member

Example - Setter

```
class Student{
        …
        int rollNo;
public:
        void setRollNo(int aRollNo){
                rollNo = aRollNo;
        }
};
```

Avoiding Error

```
void Student::setRollNo(int aRollNo){
        if(aRollNo < 0){
          rollNo = 0;
        }
        else
        {
          rollNo = aRollNo;
        }
}
```

Example - Getter

```
class Student{
```

```
        …
        int rollNo;
public:
        int getRollNo(){
                return rollNo;
        }
};
```

Good Practice:
Never return a handle to a data member from getter function because you are never sure that function accessing the reference will not change the value of the variable.

09.6.this Pointer

Consider the code of a general class given below,

```
class Student{
        int rollNo;
        char *name;
        float GPA;
public:
        int getRollNo();
        void setRollNo(int aRollNo);
…
};
```

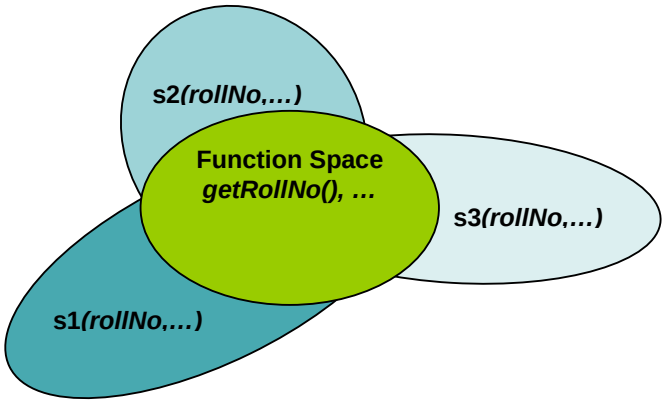
The compiler reserves space for the functions defined in the class



- Space for data is not allocated (since no object is yet created)

this Pointer
Student s1, s2, s3;

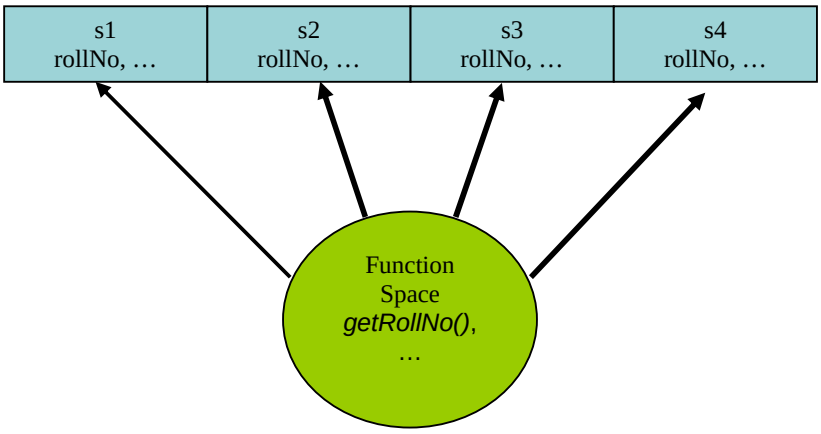
© Virtual University of Pakistan



this Pointer

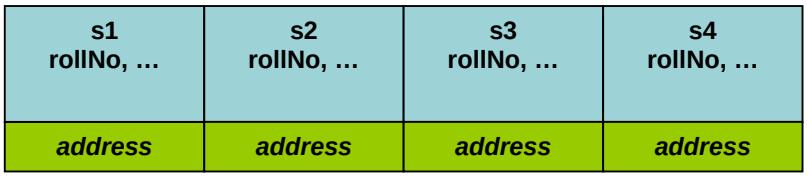
- Function space is common for every variable
- Whenever a new object is created:
  - Memory is reserved for variables only
  - Previously defined functions are used over and over again

Memory layout for objects created:



this Pointer

- Address of each object is passed to the calling function.
- This address is de-referenced by the functions and hence they act on correct objects



Passing this Pointer

- Whenever a function is called the this pointer is passed as a parameter to that function.
- Function with n parameters is actually called with n+1 parameters

Example

void Student::setName(char *)

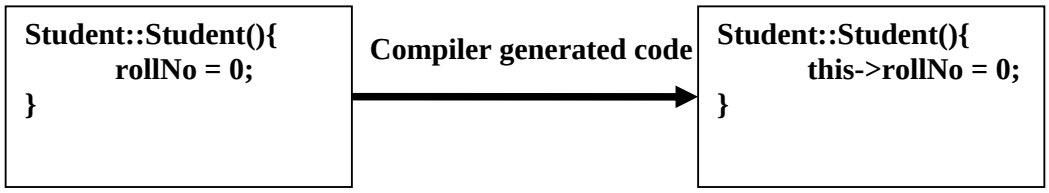
is internally represented as,

void Student::setName(char *, const Student *)

Declaration of this

DataType * const this;

Compiler Generated Code



© Virtual University of Pakistan

Lecture No.10

10.1.Uses of this Pointer

- There are situations where designer wants to return reference to current object from a function
- In such cases reference is taken from this pointer like (*this)

Example

```
Student Student::setRollNo(int aNo)
{
        …
        return *this;
}
Student Student::setName(char *aName)
{
        …
        return *this;
}
```

Usage:

```
int main()
{
        Student aStudent;
        Student bStudent;

        bStudent = aStudent.setName("Ahmad");
        …
        bStudent = aStudent.setName("Ali").setRollNo(2);

        return 0;
}
```

10.2.Separation of interface and implementation

- Public member functions exposed by a class are called interface.
- Separation of implementation from the interface is good software engineering.

Benefits of separating interface and implementation:

Consider the example of following complex no. class, this complex no. class two forms of implementations one is new and one is old implementation you can observe that if you have separated interface and implementation then we can easily change implementation without changing interface,

10.3.Complex Number

- There are two representations of complex number
  - Euler form

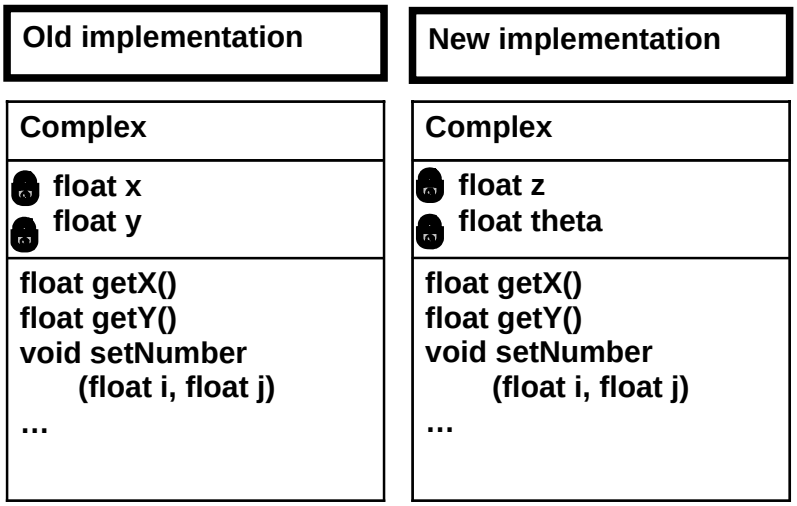
$z = x + i\,y$

  - Phasor form

$z = |z| (\cos\theta + i \sin\theta)$

z is known as the complex modulus and θ is known as the complex argument or phase

Example



- Uml notation to show private data members

Example

```
class Complex{   //old
        float x;
        float y;
public:
        void setNumber(float i, float j){
                x = i;
                y = j;
        }
        …
};
```

Example

```
class Complex{ //new
        float z;
        float theta;
public:
        void setNumber(float i, float j){
```

```
            theta = arctan(j/i);
            …
        }
        …
};
```

Advantages

1. User is only concerned about ways of accessing data (interface)
2. User has no concern about the internal representation and implementation of the class

Separation of interface and implementation

In c++ generally we can relate the concept of interface of a class to its header (.h) file and and implementation of a class to its (.cpp) file. However it is not complete separation of interface and implementation.

- Usually functions are defined in implementation file (.cpp) while the class definition is given in header file (.h)
- Some authors also consider this as separation of interface and implementation

Student.h

```
class Student{
        int rollNo;
public:
        void setRollNo(int aRollNo);
        int getRollNo();
        …
};
```

Student.cpp

```
#include "student.h"

void Student::setRollNo(int aNo){
        …
}
int Student::getRollNo(){
…
}
```

We only need to include header (.h) file in main.cpp to use the Student class as shown below,

```
Main.cpp  (main file to run the program)
#include "student.h"

int main(){
        Student aStudent;
         return 0;
}
```

10.4.const Member Functions

Some functions in our programs are general purpose functions to show or access data, they are supposed to do read only tasks only however there are chances that they can change the state of data members of the class while accessing the data members due to programming mistake, c++ provides the solution of this problem using constant member functions.

We make those functions as constant who need only read only access (for example such functions that will only display data or will return the value of data members). When we make them constant compiler generates an error if these functions try to change the value of data members of the class.

const Member Functions

Keyword const is placed at the end of the parameter list to make any function as constant.

Declaration:
Inside class

```
class ClassName{
        ReturnVal Function() const;
};
```

Definition:
Outside class

```
ReturnVal ClassName::Function() const{
        …
}
```

Example

```
class Student{
public:
        int getRollNo() const {
                return rollNo;
        }
};
```

const Functions

- Constant member functions cannot modify the state of any object

© Virtual University of Pakistan

- They are just "read-only"
- Errors due to typing are also caught at compile time

Example

Consider the function given below that is being used to check if roll no is equal to entered value if in this function we replace comparison statemtn == with assignment = statement it will compile correctly but whole code logic will change and we will get un expected result,

```
bool Student::isRollNo(int aNo){
        if(rollNo = = aNo){
                return true;
        }
        return false;
}
```

Example

```
bool Student::isRollNo(int aNo){
        /*undetected typing mistake*/
        if(rollNo = aNo){
                return true;
        }
        return false;
}
```

But if we have implemented as constant then compiler will catch this error and will produce compile time error as shown below,

Example

```
bool Student::isRollNo
                                        (int aNo)const{
        /*compiler error*/
        if(rollNo = aNo){
                return true;
        }
        return false;
}
```

const Functions

Constructors and Destructors cannot be **const because** Constructors and destructors are used to modify the object to a well defined state or to clean the memory occupied by the object.

Example

```
class Time{
public:
        Time() const {}  //error…
        ~Time() const {} //error…
};
```

const Function

- Constant member function cannot change data member
- We cannot call non constant functions in constant functions because non constant member functions may have code for changing state of the object that is not allowed in the constant functions.

Example

```
class Student{
        char * name;
public:
        char *getName();
        void setName(char * aName);
        int ConstFunc() const{
                name = getName();    //error
                setName("Ahmad");//error
        }
};
```

10.5.this Pointer and const Member Function

As we know that when a class function is called an implicit this pointer is passed to tell the function about the object it has to operate same is true for constant function with the difference that it will bbe passed as constant pointer to const data in case of constant member functions so that this pointer can not be used now to change the value of data members of the object,

```
const Student *const this;
// In case of constant member functions

instead of

Student * const this;
// In case of ordinary member functions
```

 © Virtual University of Pakistan

Lecture No.11

11.1.Usage example of Constant member functions

Problem:

Suppose we have requirement to change the class Student such that a student is given a roll number when the object is created and cannot be changed afterwards our existing class is given below,

Student Class

```
class Student{
…
        int rollNo;
public:
        Student(int aNo);
        int getRollNo();
        …
};
```

Solution of this problem:

We can do this by making rollNo constant so that cannot be changed once it is defined as shown below,

Modified Student Class

```
class Student{
…
        const int rollNo;
public:
        Student(int aNo);
        int getRollNo();
        …
};
```

Now there is only one issue of initializing this roll no with initial value but the problem is that we cannot set the value of roll no in constructor, as when code in constructor is executed the data member roll no has already been created and when we try to assign value to it in constructor compiler generates error,

Example

```
Student::Student(int aRollNo)
{
        rollNo = aRollNo;

 /*error: cannot modify a constant data member assignment statement not
```

```
initialization*/
}
```

Second solution is to write separate function but the problem remains same that we can't assign value to constant data member,

```
void Student::SetRollNo(int i)
{
        rollNo = i;
        /*error: cannot modify a constant data member again assignment statement not initialization */
}
```

We also know that we can only declare data members in structure or class but we cannot initialize them at the time of declaration in structure or class _____ because before executing constructor code, the class const member roll no has not got life it will get life along with other class members when constructor will be invoked so we can not assign any value to this constant member while declaring it.[6]

Solution:

so what is the solution of this problem as we can not initialize constant members while declaring them and we can not initialize them in constructor also because as soon as they go life they become constant to solve this problem C++ gives us new mechanism (syntax) for initialization of constant data members of the structure or class to resolve above mentioned issues,

11.2. Difference between Initialization and Assignment:

Initialization is assigning value along with creation of variable.

```
int i = 2;
```

Assignment is assigning value after creation.

```
int i;
i  = 7;
```

11.3. Member Initializer List

Member initialization list is used where we cannot modify the state of data members in the member functions of the class including constructor,

- A member initializer list is a mechanism to initialize data members
- It is given after closing parenthesis of parameter list of constructor

[6] In c++ **static const data members** can be initialized in class or structure as well.

© Virtual University of Pakistan

- In case of more than one member use comma separated list

Example

```
class Student{
        const int rollNo;
        char *name;
        float GPA;
public:
        Student(int aRollNo) : rollNo(aRollNo), name(Null), GPA(0.0){ // initialization
                …
        }
…
};
```

Order of Initialization

- Data member are initialized in order they are declared in the class
- Order in member initializer list is not significant at all

Example

```
class ABC{
        int x;
        int y;
        int z;
public:
        ABC();
};
ABC::ABC():y(10),x(y),z(y)
{
        …
}
/*      x = Junk value
                y = 10
                z = 10 */
```

11.4.const Objects

- Objects can be declared constant with the use of const keyword
- Constant objects cannot change their state

Example

```
int main()
```

```
{
        const Student aStudent;
        return 0;
}
```

Example

```
#include <cstdlib>
#include <iostream>

using namespace std;


class Student{
        int rollNo;
public:

   Student( ) {

        }

        int getRollNo(){
                return rollNo;
        }
};


int main(){

const Student aStudent;
        int a = aStudent.getRollNo();

        //error

  system("PAUSE");
   return EXIT_SUCCESS;
}
```

```
#include <cstdlib>
#include <iostream>

using namespace std;


class Student{
        int rollNo;
public:

   Student(int aRollNo) :
rollNo(aRollNo){

        }

        int getRollNo(){
                return rollNo;
        }
};

int main(){
        const Student aStudent(5);
        int a = aStudent.getRollNo();
        //error

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

const Objects

const objects can access only const member functions so chances of change of state of const objects once they are created are eliminated.
We make getRollNo function constant so that we can access it using constant objects,

Example

```
class Student{
```


© Virtual University of Pakistan

```
…
        int rollNo;
public:
…
        int getRollNo()const{
                return rollNo;
        }
};
int main(){
        const Student aStudent;
        int a = aStudent.getRollNo();
         return 0;
}
```

Constant member functions

Make all functions that don't change the state of the object constant
This will enable constant objects to access more member functions

11.5.Static Variables

Static variables of a class are such variables which are independent of class objects.

Lifetime of static variable is throughout the program life, if static variables are not explicitly initialized then they are initialized to 0 of appropriate type.

Example

Static variable is initialized once only throughout the program, independent of how many times the function initializing it is called,

```
void func1(int i){
        static int staticInt = i;
//initialization statement will be executed once
//only as static variables are initialized once
        cout << staticInt << endl;
}
int main(){
        func1(1);
        func1(2);
        return 0;
}
```

Output:
1
1

```
void func1(int i){

 static int staticInt;
 staticInt = i;
//assignment statement will be executed with each function call
 cout << staticInt << endl;
```

```
}
int main(){
        func1(1);
        func1(2);
        return 0;
}
```

Output:
1
2

Static Data Member

Definition

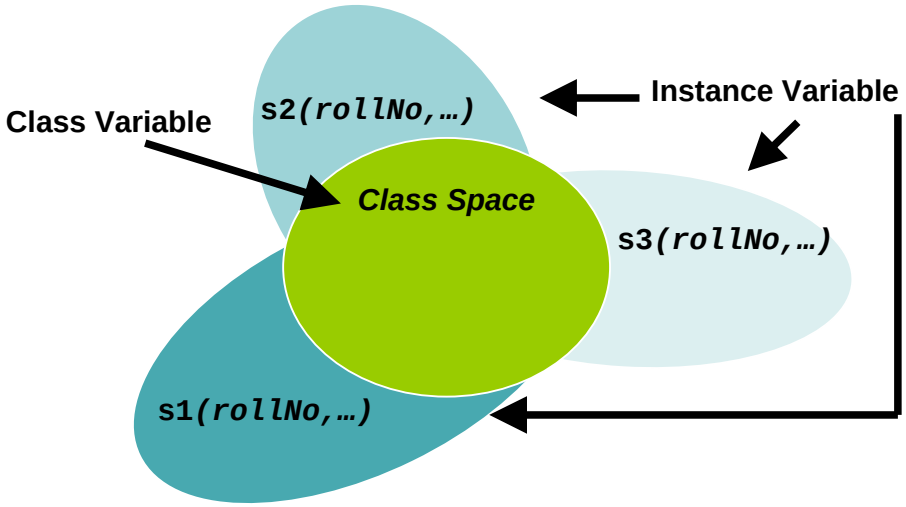
"A variable that is part of a class, yet is not part of any object of that class, is called static data member"

Static Data Member

They are shared by all instances (objects) of the class
They do not belong to any particular instance of a class

Class vs. Instance Variable

Suppose we created three objects of student class as shown below,
Student s1, s2, s3;



Static Data Member (Syntax)

Keyword static is used to make a data member static

```
class ClassName{
…
static DataType VariableName;
};
```

Defining Static Data Member

© Virtual University of Pakistan

Static data member is declared inside the class
But they are defined outside the class

Defining Static Data Member

```
class ClassName{
…
static DataType VariableName;
};

DataType ClassName::VariableName;
```

Initializing Static Data Member

Static data members should be initialized once at file scope
They are initialized at the time of definition

Example

```
class Student{
private:
static int noOfStudents;
public:
  …
};
int Student::noOfStudents = 0;
/*private static member cannot be accessed outside the class except for initialization*/
```

Initializing Static Data Member

If static data members are not explicitly initialized at the time of definition then they are initialized to 0

Example

```
int Student::noOfStudents;
```

is equivalent to

```
int Student::noOfStudents=0;
```

Lecture No.12

Review:
Static Data Member
Definition
"A variable that is part of a class, yet is not part of an object of that class, is called static data member"

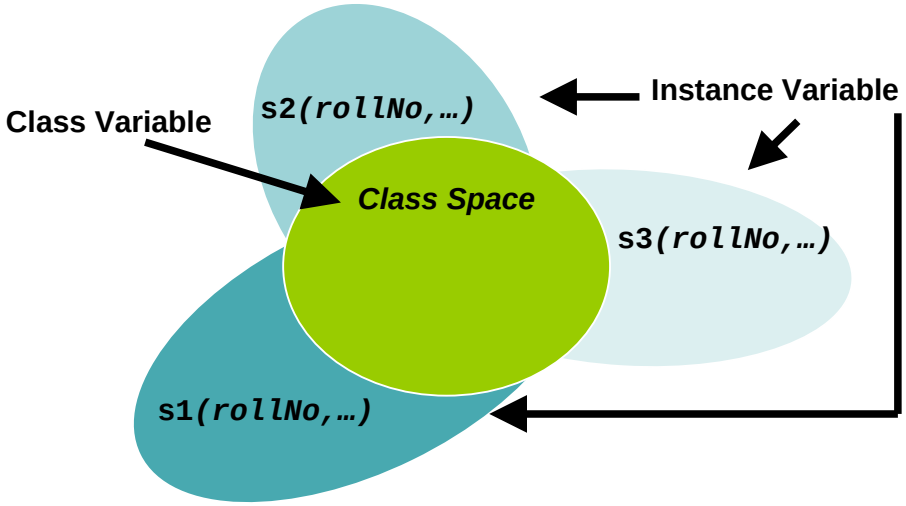
Static Data members are shared by all instances of the class and they do not belong to any particular instance of a class.

Class vs. Instance Variable

Memory for static variables in allocated in class space whereas for instance variables it is separate for each object as shown below, if we have class Student as given below,

```
class Student{
private:
static int noOfStudents;
public:
  …
};
```

When we will create objects of Student as s1, s2, s3 then memory will be allocated as given below,



Static Data Member (Syntax)
Keyword static is used to make a data member static,

```
class ClassName{
…
static DataType VariableName;
};
```


© Virtual University of Pakistan

Defining Static Data Member (allocating memory for them)

Static data member is declared inside the class
But they are defined outside the class,

```
class ClassName{
…
static DataType VariableName;
};

DataType ClassName::VariableName;
```

Initializing Static Data Member(assigning them some initial value)

Static data members should be initialized once at file scope
They are initialized at the time of definition,

Example

```
class Student{
private:
static int noOfStudents;
public:
  …
};
int Student::noOfStudents = 0;
/*private static member cannot be accessed outside the class except for initialization*/
```

Initializing Static Data Member

If static data members are not explicitly initialized at the time of definition then they are initialized to 0

Example

```
int Student::noOfStudents;
```

is equivalent to

```
int Student::noOfStudents=0;
```

12.1. Accessing Static Data Member

To access a static data member there are two ways

- Access like a normal data member (using dot operator '.')
- Access using a scope resolution operator '::'

Example

```
class Student{
public:
```

```
static int noOfStudents;
};

int Student::noOfStudents;

int main()
{
Student aStudent;
aStudent.noOfStudents = 1;
Student::noOfStudents = 1;
return 0;
}
```

12.2. Life of Static Data Member

- They are created even when there is no object of a class
- They remain in memory even when all Objects of a class are destroyed

Example

```
class Student{
public:
static int noOfStudents;
};
int Student::noOfStudents;
int main(){
Student::noOfStudents = 1;
}
```

Example

```
class Student{
public:
static int noOfStudents;
};
int Student::noOfStudents;
int main(){
{
        Student aStudent;
        aStudent.noOfStudents = 1;
}
Student::noOfStudents = 1;
return 0;
}
```

Uses

They can be used to store information that is required by all objects, like global variables

Example

Modify the class Student such that one can know the number of student created in a system

 © Virtual University of Pakistan

```
class Student{
…
public:
        static int noOfStudents;
        Student();
        ~Student();
…
};
int Student::noOfStudents = 0;
Student::Student(){
         noOfStudents++;
}
Student::~Student(){
         noOfStudents--;
}

int Student::noOfStudents = 0;
int main(){
        cout <<Student::noOfStudents <<endl;
        Student studentA;
        cout <<Student::noOfStudents <<endl;
        Student studentB;
        cout <<Student::noOfStudents <<endl;
        return 0;
}
```

Output:
0
1
2

Problem
noOfStudents is accessible outside the class
Bad design as the local data member is kept public

The solution is that we write static member function to access static members,

12.3.Static Member Function
Definition:
"The function that needs access to the members of a class, yet does not need to be invoked by a particular object, is called static member function"

- They are used to access static data members
- Access mechanism for static member functions is same as that of static data members
- They cannot access any non-static members

Example

```
class Student{
        static int noOfStudents;
        int rollNo;
public:
        static int getTotalStudent(){
                return noOfStudents;
        }
};
int main(){
        int i = Student::getTotalStudents();
        return 0;
}
```

Accessing non static data members

```
int Student::getTotalStudents(){
        return rollNo;
}
int main(){
        int i = Student::getTotalStudents();
        /*Error: There is no instance of Student, rollNo cannot be accessed*/
        return 0;
}
```

12.4.this Pointer and static member functions

- *this* pointer is passed implicitly to member functions
- *this* pointer is not passed to static member functions
- Reason is static member functions cannot access non static data members

12.5.Global Variable vs. Static Members

- Alternative to static member is to use global variable
- Global variables are accessible to all entities of the program
- User of Global variables is against the principle of information hiding.

12.6.Array of Objects

- Array of objects can only be created if an object can be created without supplying an explicit initializer
- There must always be a default constructor if we want to create array of objects

Example

```
class Test{
public:
};
int main(){
        Test array[2]; // OK
```


© Virtual University of Pakistan

```
        return 0;
}
```

```
class Test{
public:
        Test();
};
int main(){
        Test array[2]; // OK
        return 0;
}
```

```
class Test{
public:
        Test(int i);
};
int main(){
        Test array[2]; // Error
        return 0;
}
```

```
class Test{
public:
        Test(int i);
};
int main(){
        Test array[2] = {Test(0),Test(0)};
        return 0;
}
```

```
class Test{
public:
        Test(int i);
};
int main(){
        Test a(1),b(2);
        Test array[2] = {a,b};
        return 0;
}
```

Lecture No.13

13.1.Pointer to Objects

- Pointer to objects are similar as pointer to built-in types
- They can also be used to dynamically allocate objects

Example

```
class Student{
…
public:
        Student();
        Student(char * aName);
        void setRollNo(int aNo);
};
```

Example

```
int main(){
        Student obj;
        Student *ptr;
        ptr = &obj;
        ptr->setRollNo(10);
        return 0;
}
```

Allocation with new Operator

- new operator can be used to create objects at runtime

Example

```
int main(){
        Student *ptr;
        ptr = new Student;
        ptr->setRollNo(10);
        return 0;
}
```

Example

```
int main(){
        Student *ptr;
        ptr = new Student("Ali");
        ptr->setRollNo(10);
        return 0;
}
```

Example

```
int main()
{
        Student *ptr = new Student[100];
        for(int i = 0; i < 100;i++)
        {
                ptr->setRollNo(10);
        }
        return 0;
}
```


© Virtual University of Pakistan

13.2.Breakup of new Operation

new operator is decomposed as follows

- Allocating space in memory
- Calling the appropriate constructor

13.3.Case Study

Design a class date through which user must be able to perform following operations

- Get and set current day, month and year
- Increment by x number of days, months and year
- Set default date

Attributes

Attributes that can be seen in this problem statement are

- Day
- Month
- Year
- Default date

Attributes

The default date is a feature shared by all objects

- This attribute must be declared a static member

Attributes in Date.h

```
class Date
{
        int day;
        int month;
        int year;
        static Date defaultDate;
…
};
```

Interfaces

- getDay
- getMonth
- getYear
- setDay
- setMonth
- setYear
- addDay
- addMonth
- addYear
- setDefaultDate

Interfaces

As the default date is a static member the interface setDefaultDate should also be declared static

Interfaces in Date.h

```
class Date{
…
public:
        void setDay(int aDay);
        int getDay() const;
        void addDay(int x);
        …
```

```
…
};
```
Interfaces in Date.h
```
class Date{
…
public:
        static void setDefaultDate(
int aDay,int aMonth, int aYear);
…
};
```
Constructors and Destructors in Date.h
```
        Date(int aDay = 0,
  int aMonth= 0, int aYear= 0);

        ~Date(); //Destructor
};
```
Implementation of Date Class

The static member variables must be initialized

```
Date Date::defaultDate (07,3,2005);
```
Constructors
```
Date::Date(int aDay, int aMonth,
                                int aYear)    {
        if(aDay==0) {
                this->day = defaultDate.day;
        }
        else{
                setDay(aDay);
        }
        //similarly for other members
}
```
Destructor

We are not required to do any house keeping chores in destructor

```
Date::~Date
{
}
```
Getter and Setter
```
void Date::setMonth(int a){
        if(a > 0 && a <= 12){
                month = a;
}
int getMonth() const{
        return month;
}
```
addYear
```
void Date::addYear(int x){
        year += x;
        if(day == 29 && month == 2
                &&      !leapyear(year)){
                day = 1;
```

 © Virtual University of Pakistan

```
                month = 3;
        }
}
```

Helper Function

```
class Date{
…
private:
        bool leapYear(int x) const;
…
};
```

Helper Function

```
bool Date::leapYear(int x) const{
        if((x%4 == 0 && x%100 != 0)
                ||      (x%400==0)){
                return true;
        }
        return false;
}
```

setDefaultDate

```
void Date::setDefaultDate(
        int d, int m, int y){
        if(d >= 0 && d <= 31){
                day = d;
        }
        …
}
```

13.4.Complete code of Date class

```
# include<iostream.h>
# include<conio.h>
class Date{

   private:
     int day, month, year;
         static Date defaultDate;
    public:
        void setDefaultDate(int aDay,int aMonth, int aYear);
         void setDay(int aDay);
         int getDay() const;
         void addDay(int x);
         void setMonth(int aMonth);
         int getMonth() const;
         void addMonth(int x);
         void setYear(int aYear);
         int getYear() const;
         void addYear(int x);
         bool leapYear(int x)const;
         Date(int aDay , int aMonth, int aYear);
         void setDate(int aDay , int aMonth, int aYear);
```

```
        ~Date(); //Destructor
    };

    Date Date::defaultDate(07,3,2005);

Date::Date(int aDay, int aMonth, int aYear)
{
        if(aDay==0)
        {
            this->day = defaultDate.day;
        }
        else
        {
           setDay(aDay);
        }

        if(aMonth==0)
        {
            this->month = defaultDate.month;
        }
        else
        {
           setMonth(aMonth);
        }

        if(aYear==0)
        {
            this->year = defaultDate.year;
        }
        else
        {
           setYear(aYear);
        }

}

void Date::setMonth(int a) {
   if(a > 0 && a <= 12)
   {
      month = a;
   }
}

 int Date:: getMonth() const {
      return month;
   }

void Date::addYear(int x)
{
   year += x;
```


© Virtual University of Pakistan

```
    if(day == 29 && month == 2 &&  !leapYear(year))
    {
         day = 1;
         month = 3;
    }
}

bool Date::leapYear(int x) const {
    if((x%4 == 0 && x%100 != 0) || (x%400==0))
    {
          return true;
    }
    return false;
}

void Date::setYear(int aYear){
   year=aYear;
   }
void Date::setDay(int aDay){
   day=aDay;

   }

void Date::setDate(int aDay , int aMonth, int aYear){
   setDay(aDay);
   setMonth(aMonth);
   setYear(aYear);
   cout<<day<<"/"<<month<<"/"<<year<<endl;

   }

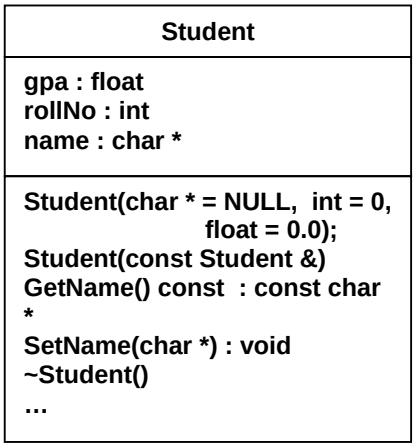

  Date::~Date(){
    cout<<"Date destructor"<<endl;
}
int main()
{
   Date aDate(0,0,0);

   aDate.setDate(20,10,2011);
   system("pause");
}
```

Lecture No.14

14.1.Composition

Consider the following implementation of the student class we discussed in previous lectures,



Composition

If one object is part of another object (relationship of part and whole) in composition lifetime of one object depends upon the other. The part objects are essential components of the whole.
For example person is **composed of** hands, eyes, feet so on.

In student class we assigning dynamic memory for variable name using new operator as shown,

```
class Student{
private:
  float gpa;
  char * name;
  int rollNumber;
public:
  Student(char * = NULL, int = 0, float = 0.0);
  Student(const Student & st);
  const char * GetName() const;
  // never return handle of private data members or private member functions
  ~Student();
  …
};

Student::Student(char * _name, int roll, float g)
{
        cout << "Constructor::Student..\n";
```


© Virtual University of Pakistan

```
        if (!_name){
                name = new char[strlen(_name)+1];
                strcpy(name,_name);
        }
        else name = NULL;
        rollNumber = roll;
        gpa = g;
 }


Student::Student(const Student & st){
  if(str.name != NULL){
     name = new char[strlen(st.name) + 1];
     strcpy(name, st.name);
  }
  else name = NULL;
  rollNumber = st.roll;
  gpa = st.g;
}


 const char * Student::GetName(){
    return name;
 }

 // never return handle of private data members or private member functions const ensures that private data members will not be changed


 Student::~Student(){
        delete [] name; // deleting name array
 }
```

In C++ *"it is all about code reuse"*

Composition is Creating objects of one class inside another class

"Has a" relationship:

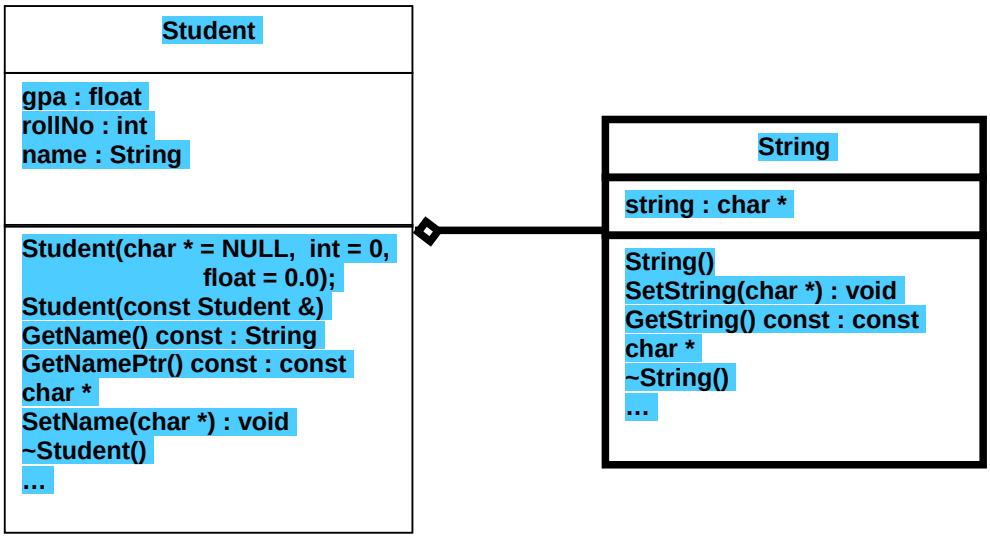
Bird has a beak

Student has a name

Composition

Now we change code slightly replacing name char * by String so that it is whole object of class String as it qualifies to be an object because we have to apply many operations on it like string dynamic creation and deletion, string copy using deep copy, searching a substring and so on….

Conceptual notation:



Composition

Now we see string class code to see how it simplifies original Student object and how we have used composition here,

```
class String{
  private:
          char * ptr;
  public:
          String(); // default constructor
          String(const String &); // copy constructor
          void SetString(const char *); // setter function
          const char * GetString() const;
// getter function returning const pointer to data member ptr
          ~String()
          …
};
 String::String(){
          cout << "Constructor::String..\n";
          ptr = NULL;
 }

 String::String(const String & str){
    if(str.ptr != NULL){
           ptr = new char[strlen(str.ptr)+1];
           strcpy(ptr, str.ptr);
    }
    else ptr = NULL;
 }
 void String::SetString(const char * str){
          if(ptr != NULL){
                   delete [] ptr;
```


© Virtual University of Pakistan

```
            ptr = NULL;
        }
        if(str != NULL){
            ptr = new char[strlen(str)+1];
            strcpy(ptr, str);
        }
 }
```

issue of memory leakage (inaccessible memory)
if we simply set pointer here
memory will be outside of our object many cause problems later
user still has pointer of passed value it can itself modify it

We resolve these two issues in SetString by allocating new memory and deleting previous memory.

```
const char * String::GetString()const{
   return ptr;
}
String::~String(){
        delete [] ptr;
        cout <<"Destructor::String..\n";
}
```

Make pointer equal to NULL as well any where you delete dynamic memory.

Now consider the code of Student class again, now by adding composed string object our code has been simplified very much, (we will use methods of composed object simply by calling them where needed)

```
class Student{
private:
 float gpa;
 int rollNumber;
 String name;
public:
 Student(char* =NULL, int=0,float=0.0);
 Student(const Student &);
 void SetName(const char *);
 String GetName() const;
 const char * GetNamePtr() const;
 ~Student();
 …
};

Student ::Student(char * _name,      int roll, float g){
        cout <<"Constructor::Student..\n";
        name.SetString(_name);
```

```
        rollNumber = roll;
        gpa = g;
 }

Student::Student(const Student & s){
        name.SetString(s.name.GetString());
        gpa = s.gpa;
        rollNumber = s.rollNumber;
 }
```

Explanation:

1. **name.SetString**(s.name.GetString());
   // setting composed name of newly created object

2. name.SetString(**s.name**.GetString());
   //accessing the composed object string name of object to be copied

3. name.SetString(s.**name.GetString**());
   //accessing the value of composed object string name by calling its member function GetString

4. **name.SetString(s.name.GetString());**
   //overall result : the value of composed object string of object to be copied will be copied to newly created object composed object string.

```
const char * Student::GetNamePtr() const{
    return name.GetString();
 }

 void Student::SetName(const char * n){
        name.SetString(n);
 }

 Student::~Student(){
        cout <<"Destructor::Student..\n";
 }

void main(){
        Student *aStudent=new Student("Fakhir", 899, 3.1);
        cout << endl;
        cout << "Name:" << aStudent->GetNamePtr() << "\n";
}
```

Output:

```
Constructor::String..
Constructor::Student..
```


© Virtual University of Pakistan

```
Name:  Fakhir
Destructor::Student..
Destructor::String..
```

Important Points:

1. We can access methods of composed object in the same way as we can access methods of other objects.

   Name of composed object.MemberFunction

2. Member functions of a class can access its private data members like,

```
Student::Student(const Student & s){
        name.SetString(s.name.GetString());
        // accessing private member String name of student using its object s
        and then accessing String name member function GetString to access
        string value two methods calss in one line
            gpa = s.gpa;
        // accessing private members of student in student member function
            rollNo = s.rollNo;
        // accessing private members of student in student member function
}
```

Constructors & Composition

Constructors of the sub-objects are always executed before the constructors of the master class

Example:

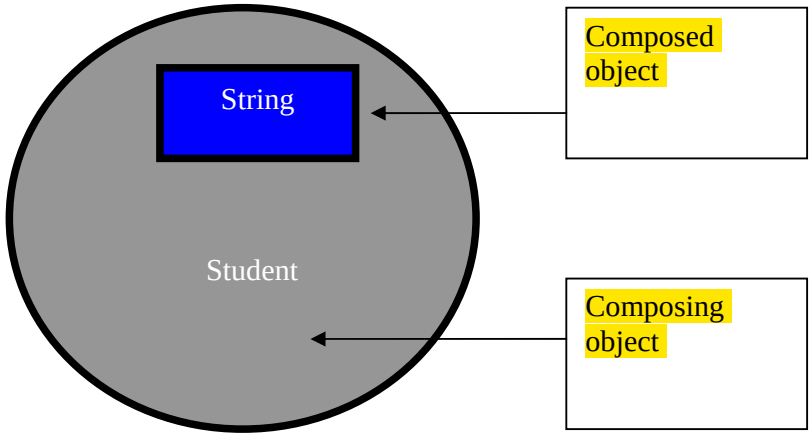
As you see the example output of program given above,

Output:

```
Constructor::String..
Constructor::Student..

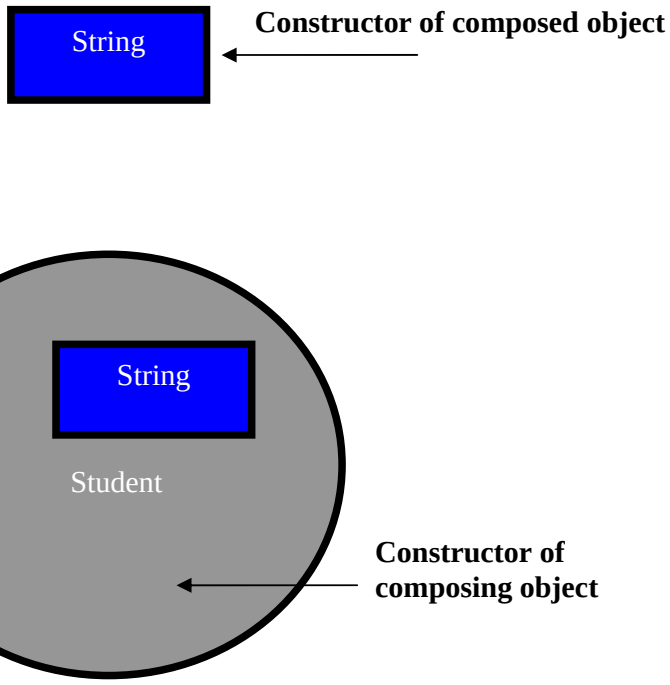
Name:  Fakhir
Destructor::Student..
Destructor::String..
```

Constructor for the sub-object name is executed before the constructor of Student and destructor of sub-object is called after destructor of student. It is logical as composing object has to contain composed object so composed object should be created first and then composing object. Similarly while destructing objects we composing object is destructed first and then composed object as shown in diagram below,



Composition

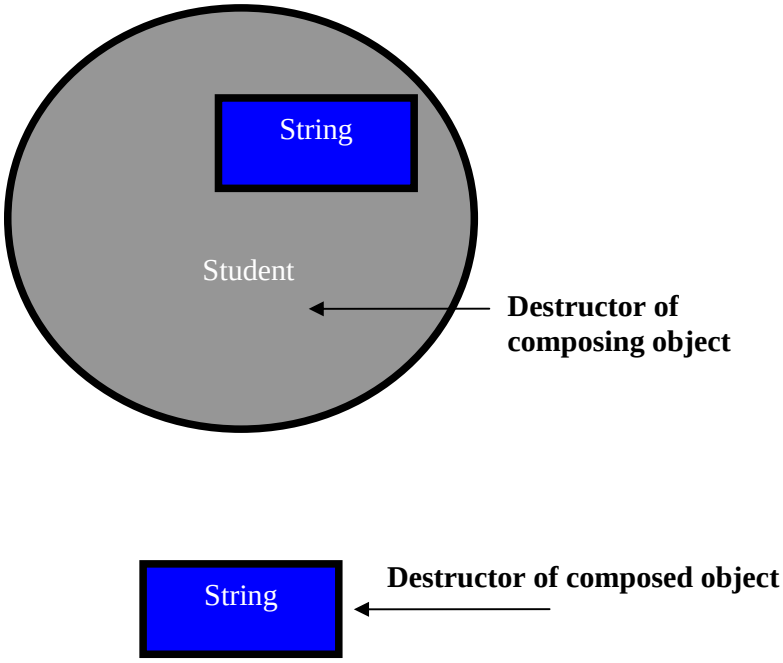
Constructor calling:
Constructors are called from composed objects to composing objects.



© Virtual University of Pakistan

Destructor calling:

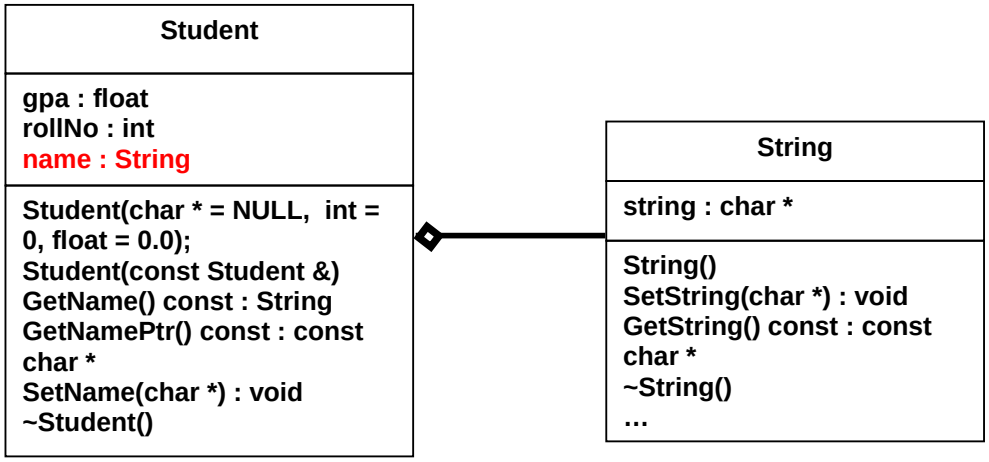
Destructors are called from composing objects to composed objects.



Lecture No.15

Composition:

We saw composition in last lecture, its Conceptual notation is given below,



We created student object in main as by passing name, rollno and gpa and then displayed the name of student using GetNamePtr member function of student class,

```
int main(){
 Student aStudent("Fakhir", 899,3.1);
 cout << endl;
 cout << "Name:" << aStudent.GetNamePtr()<< endl;
 return 0;
 }
```

Output:

The output of our code is given below,

```
Constructor::String..
Constructor::Student..
Name:  Fakhir
Destructor::Student..
Destructor::String..
```

Constructor Code:

Let us see the constructor code again,

```
Student::Student(char * n, int roll, float g){
      cout <<"Constructor::Student..\n";
      name.SetString(n);
      rollNumber = roll;
      gpa = g;
    }
```


© Virtual University of Pakistan

In this code we are setting string data member **name** of Student class using **SetString** but the problem in this approach is that we have to call SetString method explicitly to set value in string class, the reason for it is that our String class doesn't support setting its value while creating its object this is the reason we have to use the function SetString in the constructor.
This is an overhead and also not very good way to set any object value, we want to initialize our string sub-object **name** in the **student** class as we initialize other objects using constructor. For achieving this functionality we add an overloaded constructor in the **String** class that takes char string as parameter and initialize the String class object with this value using the Student constructor's "*Member initialization list*" as shown below in bold text:

```
class String{
        char *ptr;
public:
        String();
        String(char *); // constructor with char * as parameter
        String(const String &);
        void SetName(char *);
        ~String();
        …
};


String::String(char * str){
 if(str != NULL){
                ptr = new char[strlen(str)+1];
                strcpy(ptr, str);
}

else ptr = NULL;
   cout << "Overloaded Constructor::String..\n";

}
```

Now Student class constructor code is modified as follows:

```
class Student{
private:
        float gpa;
        int rollNumber;
        String name;
public:
        …
        Student(char *=NULL, int=0, float=0.0);
};
```

```
Student::Student(char * n,int roll, float g):  name(n)  {

        cout << "Constructor::Student..\n";
        rollNumber = roll;
        gpa = g;

 }

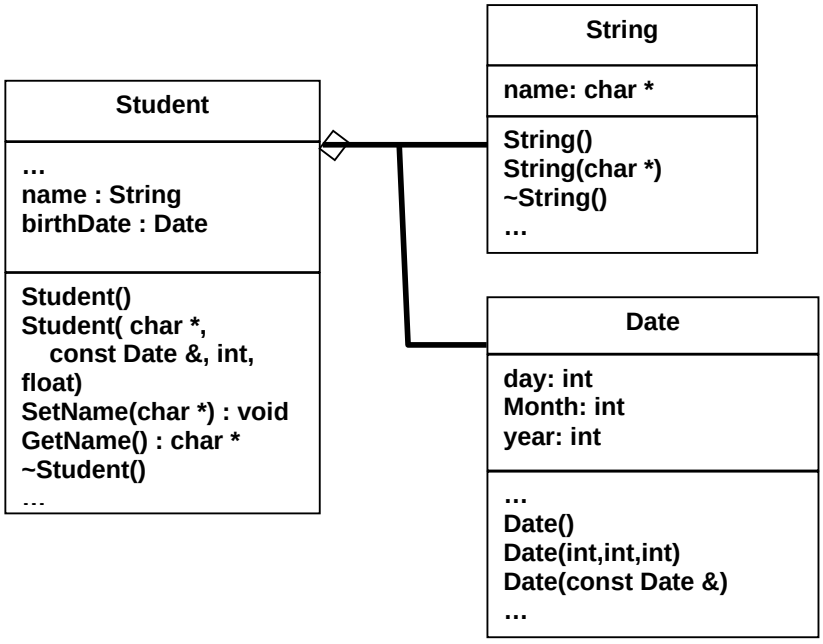
int main(){
        Student aStudent("Fakhir", 899, 3.1);
        cout << endl;
        cout << "Name:" << aStudent.GetNamePtr()          << endl;
        return 0;
}
```

Output:

```
Overloaded Constructor::String..
Constructor::Student..

Name:  Fakhir
Destructor::Student..
Destructor::String..
```

Now suppose we want to add date object in student class to store student **Birth Date,** the conceptual diagram will be as given below,



 © Virtual University of Pakistan

Student class is modified as follows:

```
class Student{
private:
        …
        Date birthDate;
        String name;
public:
        Student(char *, const Date &, int, float);
        ~Student();
        …
};
```

Composition

```
 Student::Student(char * n, const Date & d, int roll, flaot g): name(n),birthDate(d) {
        cout << "Constructor::Student..\n";
        rollNumber = roll;
        gpa = g;
 }

 Student::~Student(){
        cout << "Destructor::Student..\n";
 }
int main(){
        Date _date(31, 12, 1982);
        Student aStudent("Fakhir",   _date,899,3.5);
        return 0;
}
```

Output:

```
        Overloaded Constructor::Date..
        Copy Constructor::Date..
        Overloaded Constructor::String..
        Constructor::Student..
        Destructor::Student..
        Destructor::String..
        Destructor::Date..
        Destructor::Date..
```

15.1.Aggregation

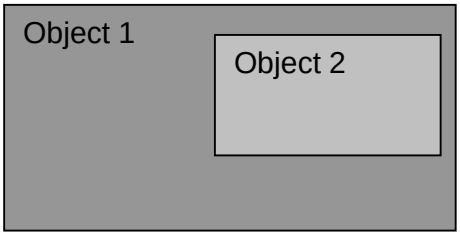
In composition we made separate object of those concepts that we think were worthy to be implemented as an object within other object to make our code simpler and to make functionality modular (divided in parts) and understandable like we made String class in Student class, but in real life most situations are such that two distinct (different) objects and one object is using services of the other one like student and teacher, student and librarian, room and chair, passenger and bus, book and

bookshelf, person and computer so on. In this case we cannot make one object as part of other object because they exist independently and only provide services to each other like in case of,

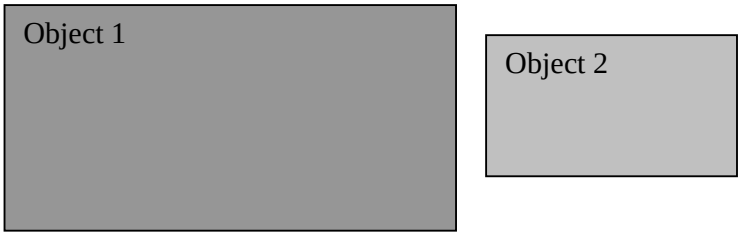
Student and Teacher: Student or Teacher object cannot be composed of other one yet they are taking services of each other.
Passenger and Bus: Passenger and Bus are taking services of each other but exist standalone also, bus includes passengers but passenger are not Part of Bus they can exist independently as well.

Composition vs. Aggregation

Aggregation is a *weak relationship* than composition because in this relationship two classes get services of each other but can exist independently as well, main difference is memory organization of two objects as shown below,



Composition

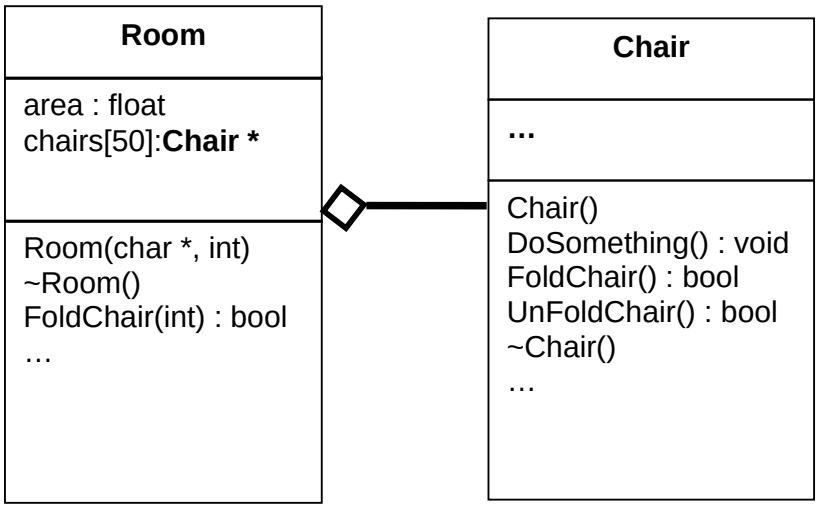


Aggregation

© Virtual University of Pakistan

Example:

Take the example of Room and Chair as given below,



Aggregation C++ implementation:

In aggregation, a pointer or reference to an object is created inside a class. The sub-object has a life that is **NOT** dependant on the life of its master class.
e.g

- Chairs can be moved inside or outside at anytime
- When Room is destroyed, the chairs may or **may not** be destroyed

Aggregation:

```
class Room{
private:
        float area;
        Chair * chairs[50];
Public:
        Room();
        void AddChair(Chair *, int chairNo);
        Chair * GetChair(int chairNo);
        bool FoldChair(int chairNo);
        …
};

Room::Room(){
for(int i = 0; i < 50; i++)
chairs[i] = NULL;
```

```
}
void Room::AddChair(Chair * chair1, int chairNo){
        if(chairNo >= 0 && chairNo < 50)
                chairs[chairNo] = chair1;
}

Chair * Room::GetChair(int chairNo){
        if(chairNo >= 0 && chairNo < 50)
                return chairs[chairNo];
        else
                return NULL;
}

bool Room::FoldChair(int chairNo){
 if(chairNo >= 0 && chairNo < 50)
        return chairs[chairNo]->FoldChair();
 else
        return false;
}
int main(){
  Chair ch1;
  {
    Room r1;
          r1.AddChair(&ch1, 1);
          r1.FoldChair(1);
  }
   ch1.UnFoldChair(1);
  return 0;
}
```

15.2.Friend Functions

The functions which are not member functions of the class yet they can access all private members of the class are called friend functions.

Why they are needed?

They are needed in situations where we have written code for some function in one class and it need to be used by other classes as well for example,
Suppose we wrote the code to compute a complex mathematical formulae in one class but later it was required by other classes as well, in that case we will make that function friend of all other classes.

Are friend functions against the concept of Object Oriented Programming?

It can be said that friend functions are against the principle of object oriented programming because they violate the principle of encapsulation which clearly says that each object methods and functions should be encapsulated in it. But there we are making our private member accessible to other outside functions.

 © Virtual University of Pakistan

Consider the following class:

```
class X{
private:
        int a, b;
public:
        void MemberFunction();
        …
};
```

Suppose we have a global function **DoSomething** that need to access the private members of class X, when we will try to access them compiler will generate error as outside world can not access private members of a class except its member functions.

```
void DoSomething(X obj){

                obj.a = 3; //Error
                obj.b = 4; //Error
        }
```

Friend Functions

In order to access the member variables of the class, we must make function friend of that class,

```
class X{
private:
  int a, b;
public:
        …
   friend void DoSomething(X obj);
};
```

Now the function **DoSomething** can access data members of class X

```
void DoSomething(X obj){

                obj.a = 3;
                obj.b = 4;
        }
```

Friend Functions

Prototypes of friend functions appear in the class definition.
But friend functions are NOT member functions.

Friend Functions

Friend functions can be placed anywhere in the class without any effect
Access specifiers don't affect friend functions or classes

```
class X{
        ...
private:
        friend void DoSomething(X);
public:
        friend void DoAnything(X);
         ...
};
```

Friend Functions

While the definition of the friend function is:

```
void DoSomething(X obj){

        obj.a = 3;              // No Error
        obj.b = 4;              // No Error
        …
}
```

friend keyword is not given in definition.

Friend Functions

If keyword **friend** is used in the function definition, it's a syntax error

```
//Error…

friend void DoSomething(X obj){
        …
}
```

Friend Classes

Similarly, one class can also be made friend of another class:

```
class X{
        friend class Y;
        …
};
```

Member functions of class Y can access private data members of class X

```
class X{
        friend class Y;
        private:
                int x_var1, x_var2;
          ...
        };
class Y{
        private:
                int y_var1, y_var2;
```

 © Virtual University of Pakistan

```
        X objX;
    public:
        void setX(){
            objX.x_var1 = 1;
        }
    };
int main(){
        Y objY;
        objY.setX();
        return 0;
    }
```

Lecture No.16

16.1.Operator overloading

Consider the following class,

```
class Complex{
private:
        double real, img;
public:
        Complex Add(const Complex &);
        Complex Subtract(const Complex &);
        Complex Multiply(const Complex &);
        …
};
```

We want to write function to add two complex no. objects, the Add function implementation to add two complex numbers is shown below, this function is taking one complex no object and its adding the current (with reference to which it will be called) and is returning result in new object,

```
Complex Complex::Add(const Complex & c1){
        Complex t;
        t.real = real + c1.real;
        t.img  = img  + c1.img;
        return t;
};

// adds the contents of c2 to c1
// creating new object c3 and assigning it result of c1+c2
```

Now we can add two complex no. objects using the following statement,

```
Complex c3 = c1.Add(c2);
```

In this statement two operations are taking place,

1. One is addition of two objects using the function call Add and returning the result in a new object.
   Complex c3 = **c1.Add(c2);**

2. Second is copy of that temporary object to newly created object c3 using copy constructor.
   Complex c3 = c1.Add(c2);

But there are two issues with this implementation,

1. We can't add two complex no. objects by simple writing '+' as we can add basic data types like int or float as shown below,

int a = 3;

 © Virtual University of Pakistan

```
int b = 5;
c = a + b;  // correct

Complex c1(2,3), c2(4,5);
Complex c3 = c1 + c2; // error
```

Instead we have to explicitly write,

Complex c3 = c1.Add(c2)

If we give our complex no. class code some user in compiled form for use, user will need to know how we have written **Add** function (no. of parameters , return type) to add two complex no. objects so that he can call Add function correctly.

2. If we want to perform add operation on more than two objects in a single mathematical statement like:

c1+c2+c3+c4

We are unable to do it.

We have to explicitly write,

c1.Add(c2.Add(c3.Add(c4)))

Alternative way is:

```
t1 = c3.Add(c4);
t2 = c2.Add(t1);
t3 = c1.Add(t2);
```

This is also overhead, especially if the mathematical expression is large,

Converting it to C++ code will involve complicated mixture of function calls
Code will become less readable
Chances of human mistakes will become very high
Code produced will be very hard to maintain

The solution to this problem is simple that we can write normal operators like +,-,*, and so on for our user defined classes as well,

It is ***"Operator overloading"***

Using operator overloading we can perform basic operations (like addition, subtraction, multiplication, division and so on…) on our own defined classes objects in the similar way as we perform them on basic built-in types(like int, float, long, double etc.).
C++ allows us to overload common operators like **+**, **-** or * etc…
With operator overloading Mathematical statements don't have to be explicitly converted into function calls as we had to do to add two complex no objects using function call **Add**.

Operator overloading

Assume that operator **+** has been overloaded then actual C++ code becomes:

c1+c2+c3+c4

The resultant code is very easy to read, write and maintain

Operator overloading

C++ automatically overloads operators for pre-defined types as these have also been implemented as classes by c++.

Example of predefined types:

int
float
double
char
long

Operator overloading

```
float x;
int y;
x = 102.02 + 0.09; // overloaded operator '+' for float type will be called by c++
Y = 50 + 47; // overloaded operator '+' for int type will be called by c++
```

The compiler probably calls the correct overloaded low level function for addition i.e:

```
// for integer addition:
Add(int a, int b)

// for float addition:
Add(float a, float b)
```

Operator overloading

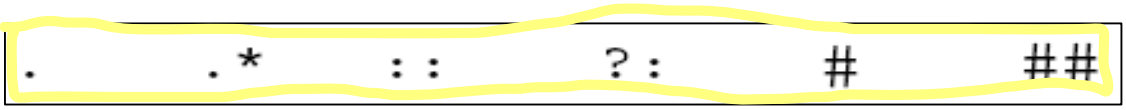
Operator functions are not usually called directly, they are automatically invoked to evaluate the operations they implement by compiler.

List of operators that can be overloaded in C++:

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

List of operators that can't be overloaded:

© Virtual University of Pakistan



Reason: They take actual current object name, rather than value in their argument as you have seen previously in the use of dot (**'.'**) operator,

Student std;
int roll = std.getRollNo() // dot operator is performing on actual function (getRollNo) of class Student that will vary from program to program.

?: is the only ternary operator in C++ and can't be overloaded.

The precedence of an operator:

The precedence of an operator is order of evaluation which operator will be evaluated first in expression.
The precedence of an operator is **NOT** affected due to overloading.
Example:

c1*c2+c3
c3+c2*c1
In both lines multiplication * will be done first and then addition.

Associativity:

Associativity is **NOT** changed due to overloading
Following arithmetic expression always is evaluated from left to right:
c1 + c2 + c3 + c4

Unary operators and assignment operator are right associative, e.g:
a=b=c is same as **a=(b=c)**
All other operators are left associative:
c1+c2+c3 is same as
(c1+c2)+c3

Important things to consider:
Always write code representing the operator for example adding subtraction code inside the + operator will create chaos.
Creating a new operator is a syntax error (whether unary, binary or ternary), you cannot create $.

Arity of Operators

Arity (no of operands it works on) of an operator is NOT affected by overloading
Example:
Division operator will take exactly two operands in any case:
b = c / d

General syntax of Operators Overloading:

In case of member functions of a class:

```
return_type class_name::operator operator_symbol( parameters ){
        /*code*/
}
```

In case of non member functions of a class (in this case we will make overloaded operator function as friend function):

```
return_type operator operator_symbol( parameters ){
        /*code*/
}
```

For example:

```
Complex& Complex::operator + (const Complex & c){
/*code*/
}

Complex& operator + (const Complex & c){
/*code*/
}
```

Binary Operators Overloading:

Binary operators act on two quantities.
Examples of binary operators:

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	,	->*	->		

General syntax of Binary Operators Overloading:

In case of member function of a class:

```
TYPE class_name::operator operator_symbol( TYPE rhs ){
        /*code*/
}
```

In case of non-member function of a class:

```
TYPE class_name::operator operator_symbol( TYPE rhs1, TYPE rhs2 ){
        /*code*/
}
```


© Virtual University of Pakistan

The **"operator OP"** must have at least one formal parameter of type class (user defined type)

Following is an error:
int operator + (int, int);

Examples:

Overloading + operator:

```
class Complex{
private:
        double real, img;
public:
        …
        Complex operator +(const Complex & rhs);
};

Complex Complex::operator +( const Complex & rhs){
        Complex t;
        t.real = real + rhs.real;
        t.img = img + rhs.img;
        return t;
}
```

The return type is Complex so as to facilitate complex statements like:
Complex t = c1 + c2 + c3;

The above statement is automatically converted by the compiler into appropriate function calls:
(c1.operator +(c2)).operator +(c3);

If the return type was **void**,

```
class Complex{
        ...
 public:
        void operator+(
                const Complex & rhs);
};

void Complex::operator+(const Complex & rhs){
        real = real + rhs.real;
        img = img + rhs.img;
};
```

We have to do the same operation **c1+c2+c3** as:
c1+c2
c1+c3
// final result is stored in c1

Drawbacks of void return type:

Assignments and cascaded expressions are not possible
One of the existing objects is used to store result
Code is less readable
Debugging is tough
Code is very hard to maintain

© Virtual University of Pakistan

Lecture No.17

Binary operators (cont.)

The binary operator is always called with reference to the left hand argument.

Example:
In **c1+c2**,
c1.operator+(c2)
// c1 is calling overloaded + operator and c2 is being passed as
// reference in that function.
In **c2+c1**,
c2.operator+(c1)
// c2 is calling overloaded + operator and c1 is being passed as
// reference in that function.

Adding basic data type to complex number class:

The overloading code we discussed before for complex no. class can add two complex number objects but it can not handle the following situation:

Complex c1;
c1 + 2.325 (as we are adding basic data type double to complex no.)

To do this, we have to modify the **Complex** class.

Modifying the complex class:

```
class Complex{
...
Complex operator+(const Complex & rhs);
Complex operator+(const double& rhs);
};

Complex operator + (const double& rhs){
        Complex t;
        t.real = real + rhs;
        t.img = img;
        return t;
}
```

Now we can write both forms of statements,
Complex c2, c3;
Complex c1 = c2 + c3;
Complex c4 = c2 + 235.01;
But problem arises if we do the following:

Complex c5 = 450.120 + c1;
The + operator is called with reference to **450.120**

No predefined overloaded + operator is there that takes **Complex** as an argument

Now if we write the following two functions to the class, we can add a **Complex** to a **real** or vice versa :

```
Class Complex{
…
friend Complex operator + (const Complex & lhs, const double & rhs);
friend Complex operator + (const double & lhs, const Complex & rhs);
};
```

We have made them as friend so that we can write them as non member functions and they are not called with respect to complex no. class object instead we pass both arguments (complex no. object and double value) to this function compiler invoke them according to arguments passed. Their implementation is similar,

```
Complex operator +(const Complex & lhs, const double& rhs){

        Complex t;
        t.real = lhs.real + rhs;
        t.img = lhs.img;
        return t;
}

Complex operator + (const double & lhs, const Complex & rhs){

        Complex t;
        t.real = lhs + rhs.real;
        t.img = rhs.img;
        return t;
}
```

Binary operators

So adding three overloaded versions for + operator as shown below allow us to write code to,
Add two complex objects
Add complex object and a double value.

```
Class Complex{
…
Complex operator + (const Complex &);
friend Complex operator + (const    Complex &, const double &);
friend Complex operator + (const    double &, const Complex &);
};
```

Non members which are not friend call also achieve this functionality but in that case we need extra four functions two getters and two setters each for real and imag part.
Compiler searches overloaded operator code in member function first and then in non member functions.

 © Virtual University of Pakistan

Binary operators

Other binary operators are overloaded in similar to the + operator as demonstrated in the above examples
Example:

```
Complex operator * (const Complex & c1, const Complex & c2);
Complex operator / (const Complex & c1, const Complex & c2);
Complex operator - (const Complex & c1, const Complex & c2);
```

17.1. Overloading Assignment operator

As we know compiler can generate the following three functions for a class if required on its own,

- Default Constructor (in case we have not written any other constructor for a class)
- Copy Constructor
- Assignment Operator

However as we discussed previously if our class has any data member using dynamic memory then we had to write our own code for default constructor, copy constructor and similarly assignment operator as compiler generated version of these functions performs shallow copy that creates dangling pointer, and memory leakage issues in case of dynamic memory allocation.
We have already seen code of default constructor and copy constructor and the code for overloaded assignment operator is similar,

Consider the string class:

```
class String{
        int size;
        char * bufferPtr;
public:
        String(); // default constructor
        String(char *); // overloaded constructor
        String(const String &); // copy constructor
        …
};
String::String(){

                bufferPtr = NULL;
               size = 0;
}

String::String(char * ptr){
        if(ptr != NULL){
                size = strlen(ptr);
                bufferPtr = new char[size+1];
                strcpy(bufferPtr, ptr);
        }
```

```
        else{
                bufferPtr = NULL;
size = 0;
}
}

String::String(const String & rhs){
        size = rhs.size;

        if(rhs.size != 0){
                bufferPtr = new char[size+1];
                strcpy(bufferPtr, ptr);
        }
        else
                bufferPtr = NULL;

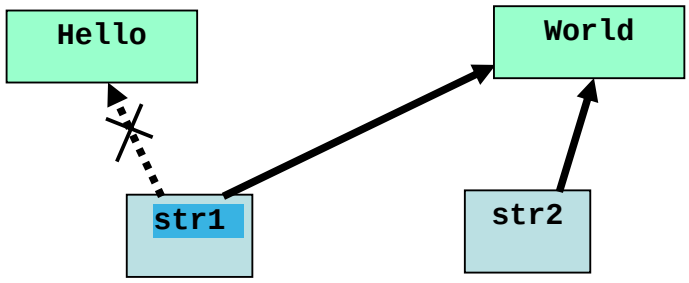
}

int main(){
        String str1("Hello");
        String str2("World");
        str1 = str2;
        return 0;
}
```

Member wise copy assignment

Assignment operator (Shallow Copy)

Result of **str1 = str2** (memory leak)



Second issue is dangling pointer issue as was in the case of copy constructor.

Modified Assignment Operator Code:

So we add overloaded assignment operator to perform deep copy as given below,

[7] Here by term member wise copy we mean copying values of members of class one by one blindly also called bitwise copy.
[8] In ANSI c++ standard term member wise copy has also been used to indicate logical copy (the deep copy)

 © Virtual University of Pakistan

```
class String{
        …
public:
        …
        void operator =(const String &);
};

void String::operator = (const String & rhs){
        size = rhs.size;
        if(rhs.size != 0){
          delete [] bufferPtr; // resolving inaccessible memory issue
          bufferPtr = new char[rhs.size+1]; // creating new dynamic memory
          strcpy(bufferPtr,rhs.bufferPtr); // deep copy
        }
        else
                bufferPtr = NULL;

}

int main(){
        String str1("ABC");
        String str2("DE"), str3("FG");
        str1 = str2;         // Valid…
        str1 = str2 = str3;  // Error…
        return 0;
}
```

The problem in statement
str1 = str2 = str3 is,

str1=str2=str3 is resolved as:
str1.operator=(str2.operator=(str3))
Assignment operator is beiing called two times one for part str2 **=** str3 and then for str1 **=** (str2 = str3) as assignment operator is right associate so first **str2=str3** will be executed, and str2 will become equal to str3, then first overloaded assignment operator execution result will be assigned to s1,
str1.operator=(**str2.operator=(str3)**)

Problem is when compiler will try to invoke second assignment operator to assign value to str1 error will be returned becuase of void return type of oveloaded assignment operator the reasn is explained below,
As we have studied before values are passed as parameters in operator overloadnig,
str2 = str3 means str2.operator = (str3)
// str3 is being called as parameter
and
str1 = str2 = str3 means str3.operator(str2.operator = (str3))
// str2.operator = (str3) is being passed as parameter

This issue can be resolved by introducing minor change in our code of copy assignment operator to make it return String object instead of void as shown below,

```
class String{
…
public:
        …
        String & operator = (const String &);
};

String & String :: operator = (const String & rhs){
        size = rhs.size;
        delete [] bufferPtr;
        if(rhs.size != 0){
          bufferPtr = new char[rhs.size+1];
          strcpy(bufferPtr,rhs.bufferPtr);
        }
        else bufferPtr = NULL;
        return *this;
}
```

Now we are returning the value by reference of the object with respect to which this overloaded assignment operator will be called. It will be str2 in the case of str2 = str3, now when part **str1 = (**str2 = str3**)** will be executed, str2 will be passed as argument, that will be assigned to str1.

© Virtual University of Pakistan

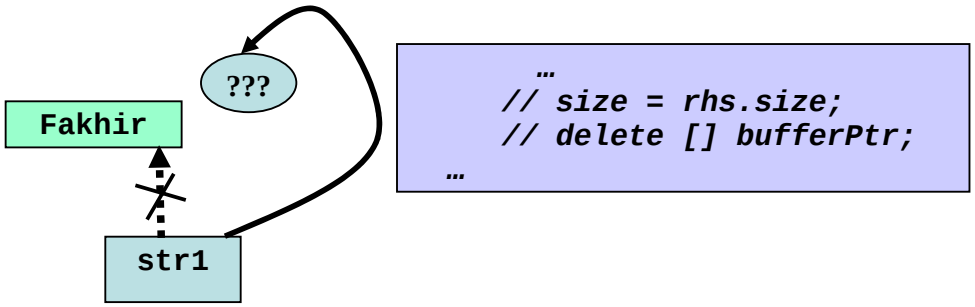
Lecture No.18

18.1.Self assignment problem:

In we assign same string to itself as done in main function below our program will produce unexpected results as source and destination operands for copying are same,

```
int main(){
        String str1("Fakhir");
        str1 = str1; // Self Assignment problem…
        return 0;
}
```

Result of str1 = str1



We can resolve this issue by adding a simple if condition to ensure that both strings are not same

```
String & String :: operator = (const String & rhs){
 if(this != &rhs){
        size = rhs.size;
        delete [] bufferPtr; // deleting memory of left hand side operand
        if(rhs.bufferPtr != NULL){
          bufferPtr = new char[rhs.size+1];
          strcpy(bufferPtr,rhs.bufferPtr);
    // memory access violation or incorrect data copy
        }
        else bufferPtr = NULL;
 }
 return *this;
}
```

Now self-assignment is properly handled:

```
int main(){
        String str1("Fakhir");
        str1 = str1;
        return 0;
}
```

We can make return type String & as constant to avoid assignment to sub expressions, like (str1 = str2) = str3

```
class String{
…
public:
        …
        const String & operator=[9]
                        (const String &);
};

int main(){
        String s1("ABC"),
                s2("DEF"),
                s3("GHI");
        // Error…
        (s1 = s2) = s3;
        return 0;
}
```

But as we can do that with primitive types so we can allow assignment to sub expressions by making return type as String & only as we have done before.

```
int main(){
        int a, b, c;
        (a = b) = c;
        return 0;
}
```

18.2.Other Binary operators

Overloading += operator:

```
class Complex{
  double real, img;
public:
Complex & operator+=(const Complex & rhs);
Complex & operator+=(const double & rhs);
...
};

Complex & Complex::operator += (const Complex & rhs){
        real = real + rhs.real;
        img = img + rhs.img;
        return * this;
}

Complex & Complex::operator += (const double & rhs){
        real = real + rhs;
        return * this;
```

[9] We have seen previously that we should not return handle to any private data member

 © Virtual University of Pakistan

```
}

int main(){
        Complex c1, c2, c3;
        c1 += c2;
        c3 += 0.087;
        return 0;
}
```

18.3.Friend Functions and Operator overloading

Friend functions minimize encapsulation as we can access private data of any class using friend functions,
This can result in:

- Data vulnerability
- Programming bugs
- Tough debugging

Hence, use of friend functions must be limited we can overload operators without declaring them friend functions of a class, for example the + operator can be defined as a non-member, non-friend function as shown below,
(Three versions of overloaded + operator for Complex no class to handle three kinds of statements)

- obj1 + obj2
  - + obj1
- obj1 + 3.78

```
Complex operator + (const Complex &  a, const Complex & b){
        Complex t = a; // creating temporary object t to store a+b
        return t += b; // returning t by reference
}

Complex operator + (const double & a, const Complex & b){
        Complex t = b;
        return t += a;
}

Complex operator + (const Complex & a, const double & b){
        Complex t = a;
        return t += b;
}
```

Other Binary operators

The operators
-=, /=, *=, |=, %=, &=, ^=, <<=, >>=, !=
can be overloaded in a very similar fashion.

Lecture No.19

Overloading stream insertion extraction operators

19.1. Stream Insertion operator

Often we need to display the data on the screen c++ provides us insertion operator ('<<')to put data on output stream default is console but it can be any file or network socket as well to send data on network from our program.

Example:

```
int i=1, j=2;
cout << "i= "<< i << "\n";
cout << "j= "<< j << "\n";
```

19.2. Stream Extraction operator

We also need to get data from the console or from file or network this is achieved through c++ provided stream extraction operator ('>>') that is used to get data from input stream, again default input stream is from console.

Example:

```
int i,j;
cin >> i >> j; // getting value of i and j from user
```

Explanation:

cin and **cout** are objects of istream and ostream classes used for input and output respectively, the insertion and extractions operators have been overloaded in istream and ostream classes to do these tasks.

When we write lines like,

```
int i;
cin>> i;
cout << i;
```

Actually we are using istream and ostream class objects and using these objects we are calling these classes overloaded (>> and <<) operators that have been overloaded for all basic types like integer, float , long , double and char *.
We have seen previously that actual call of overloaded operators for a class takes place by passing the objects to overloaded function as parameter like shown below,

```
cin>>i;
istream & operator >> (istream & in, int & i)
```

Here cin will be passed as istream object along with int i to be displayed and code of this functions is returning istream object by reference & to accommodate multiple input statement in a single line like,

```
int i , j;
cin>>i >> j;
```

as we did in our string class before.

© Virtual University of Pakistan

Same goes for insertion operator <<
cout << i;
ostream & operator >> (ostream & os, const int & i)

Stream insertion and extraction operator have been overloaded for basic data types but if we try to use them for user defined data types like our Complex no. class compiler will generate error as it will not find any overloaded operator code for our complex no class

```
Complex c1;
cout << c1;  // Error
cout << c1 << 2; // Error cascaded statement
```

// Compiler error: binary '<<' : no operator // defined which takes a right-hand operand of type 'class Complex'

Same error will be for stream extraction operator so will need to overload these two operators (<< and >>) for our Complex no. class.

19.3.Overloading Stream Insertion Operator

First we try to overload insertion << operator as member function as we did before,

```
class Complex{
        …
public:
        …
        void operator << (const
                                Complex & rhs);
};
```

But when we will call this overloaded function is main compiler will generate errors as shown below,

```
int main(){
        Complex c1;
        cout << c1;             // Error
        c1 << cout;
        c1 << cout << 2; // Error
        return 0;
};

class Complex{
        …
public:
        …
        void operator << (ostream &);
};

void Complex::operator << (ostream & os){
```

```
        os      << '('  << real
                << ',' << img << ')';
}
```

Now the statement c1 << cout will work but it has two limitations,
Difficult to understand and remember statement syntax (c1 << cout ;)
Cascaded statements not possible (cout << c1 << 2 ;)

Better syntax is given below to resolve these two issues,

```
class Complex{
  ...
friend ostream & operator << (ostream & os, const Complex & c);
};
```

Stream Insertion operator

```
// we want the output as: (real, img)
ostream & operator << (ostream & os, const Complex & c){
        os << '(' << c.real
          << ','
          << c.img << ')';
        return os;
}
```

ostream reference can not be const as it store the data in its buffer to insert on output stream, however Complex reference will be constant as we are only getting data from Complex object and inserting it to output stream.

```
Complex c1(1.01, 20.1), c2(0.01, 12.0);
cout << c1 << endl << c2;
```

Stream Insertion operator

Output:

```
( 1.01 , 20.1 )
( 0.01 , 12.0 )
```

Now cascading statements are also possible as given below,

```
cout << c1 << c2;
```

is equivalent to

operator<<(**operator<<(cout,c1)**,c2);

Because insertion operator is Left to right associative so first left part **cout << c1** << c2 will be executed and then the next part as opposed to copy assignment operator that will right associative.

Same thing can be done with stream extraction operator,

19.4. Overloading Stream Extraction Operator:

 © Virtual University of Pakistan

class Complex{
...
friend istream & operator >> (istream & i, Complex & c);
};

istream can not be cont and istream buffer will change as we will get data from it and assign it to complex reference similarly Complex object can not be const for stream extraction operator as well because we will add data to it and hence its state will change.

Stream Extraction Operator Code:

```
istream & operator << (istream & in, Complex & c){
        in >> c.real;
        in >> c.img;
        return in;
}
```

Main Program:

```
Complex c1(1.01, 20.1);
cin >> c1;
// suppose we entered    // 1.0025 for c1.real and     // 0.0241 for c1.img
cout << c1;
```

Output:

```
( 1.0025 , 0.0241 )
```

19.5.Other Binary operators:

Overloading comparison operators (Equality and Inequality operators)

```
class Complex{
public:
        bool operator == (const Complex & c);
//friend bool operator == (const //Complex & c1, const Complex & c2);
        bool operator != (const Complex & c);
//friend bool operator != (const //Complex & c1, const Complex & c2);
        …
};
```

Equality operator:

```
bool Complex::operator ==(const Complex & c){
        if((real == c.real) &&
          (img == c.img)){
```

```
                return true;
        }
        else
                return false;
}
```

As non member friend function:

```
bool operator ==(const Complex& lhs, const Complex& rhs){
        if((lhs.real == rhs.real) &&
          (lhs.img == rhs.img)){
                return true;
        }
        else
                return false;
}
```

Inequality Operator:

```
bool Complex::operator !=(const Complex & c){
        if((real != c.real) ||
          (img != c.img)){
                return true;
        }
        else
                return false;
}
```


© Virtual University of Pakistan

Lecture No.20

Modified String Class:

We have seen the following string class till now,

```
class String{
private:
        char * bufferPtr;
    int size;
public:
        String();
        String(char * ptr);
        void SetString(char * ptr);
        const char * GetString();
        ...
};

int main(){
        String str1("Test");
        String str2;
        str2.SetString("Ping");
        return 0;
}
```

What if we want to change the string from "*Ping*" to "*Pong*"?? *{ONLY 1 character to be changed…}*

Possible solution:

- Call: **str2.SetString("Pong");**
- This will delete the current buffer and allocate a new one
- Too much overhead if string is too big

Or, we can add a function which changes a character at nth location

```
class String{
        ...
public:
        void SetChar(char c, int pos);
        ...
};

 void SetChar(char c, int pos){
        if(bufferPtr != NULL){
                if(pos>0 && pos<=size)
                        bufferPtr[pos] = c;
        }
}
```

Other Binary Operators

Now we can efficiently change a single character:

```
String str1("Ping");
```

str1.SetChar('o', 2);
*// str1 is now changed to "**Pong**"*

20.1. Subscript [] Operator

There is another elegant solution present for this problem. It is subscript operator that is used on basic char [] data type as shown below,

```
char array[5] = "Ping";
array[1] = 'o';
```

We want to get same functionality for our own defined String class for this we overload the subscript **"[]"** operator.

We want function like given below for subscript operator in our String class,

```
int main(){
        String str2;
        str2.SetString("Ping");
        str[2] = 'o';
// acting as l-value (left value so that we can assign it some value)
        cout << str[2];
// acting as r-value (we are reading value using subscript operator)
        return 0;
}
```

20.2. Overloading Subscript [] Operator

Subscript operator must be overloaded as member function of the class with one parameter of integer type,

```
class String{
        ...
public:
        char & operator[](int);
        ...
};

char & String::operator[]( int pos){
        assert(pos>0 && pos<=size);
        return stringPtr[pos-1];
}
int main() {
        String s1("Ping");
        cout <<str.GetString()<< endl;
        s1[2] = 'o';
        cout << str.GetString();
        return 0;
}
```

Output:

© Virtual University of Pakistan

Ping
Pong

20.3.Overloading Function () operator

Must be a member function
Any number of parameters can be specified
Any return type can be specified
Operator() can perform any generic operation

Function Operator

```
class String{
        ...
public:
        char & operator()(int);
        ...
};

char & String::operator()
                                (int pos){
        assert(pos>0 && pos<=size);
        return bufferPtr[pos-1];
}
int main(){
String s1("Ping");
char g = s1(2); // g = 'i'
s1(2) = 'o';
cout << g << "\n";
cout << str.GetString();
return 0;
}
```

Output:

i
Pong

20.4.Function Operator performing Sub String operation,

```
class String{
        ...
public:
        String operator()(int, int);
        ...
};

String String::operator()(int index,    int subLength){
        assert(index>0 && index+subLength-1<=size);
        char * ptr = new char[subLength+1];
```

```
        for (int i=0; i < subLength; ++i)
    ptr[i] = bufferPtr[i+index-1];
        ptr[subLength] = '\0';
        String str(ptr);
        delete [] ptr;
        return str;
}

int main(){
        String s("Hello World");
        // "<<" is overloaded
        cout << s(1, 5);
        return 0;
}
```

Function Operator
Output:

```
Hello
```

20.5. Unary Operators

Unary operators take one operand, they act on the object with reference to which they have been called as shown below,
& * + - ++ -- ! ~

Examples:

- **--x**
- **-(x++)**
- **!(*ptr ++)**

Unary Operators

Unary operators are usually prefix, except for **++** and **--**
++ and **--** both act as prefix and postfix

Example:
h++;
g-- + ++h - --i;

General syntax for unary operators
As Member Functions:
TYPE & operator OP (); // no argument the object with respect to which it is called is taken as one operand
As Non-member Functions:
Friend TYPE & operator OP (TYPE & t);
// one argument object with respect to which it is called.

© Virtual University of Pakistan

Overloading unary '-':

```
class Complex{
...
Complex  operator - ( );
// friend Complex operator -(Complex &);
};

Complex Complex::operator -(){
        Complex temp;
        temp.real = -real;
        temp.img = -img;
        return temp;
}
```

Complex c1(1.0 , 2.0), c2;
c2 = -c1;
// c2.real = -1.0
// c2.img = -2.0

Unary '+' is overloaded in the same way.

Lecture No.21

Unary Operators

21.1. Behavior of ++ and -- for pre-defined types:

- Post-increment **++:**

Post-increment operator ++ increments the current value and then returns the previous value

- Post-decrement --: Works exactly like post ++

Example:

```
int x = 1, y = 2;
cout << y++ << endl;
cout << y;
```

Output:

2
3

Example:

```
int y = 2;
y++++;// Error
y++ = x;          // Error
```

Behavior of ++ and -- for pre-defined types:

- Pre-increment ++:

Pre-increment operator ++ increments the current value and then returns it's reference

- Pre-decrement --:

Works exactly like Pre-increment ++

Example:

```
int  y = 2;
cout << ++y << endl;
cout << y << endl;
```

Output:

3
3

Example:

```
int x = 2, y = 2;
++++y;
cout << y;
++y = x;
cout << y;
```

Output:

4
2

© Virtual University of Pakistan

Example (Pre-increment):

```
class Complex{
        double real, img;
public:
...
        Complex & operator ++ ();
    // friend Complex & operator ++(Complex &);
};

Complex & Complex::operator++(){       // member function
        real = real + 1;
        return * this;
}

Complex & operator ++ (Complex & h){  // non member function
        h.real += 1;
        return h;
}
```

Example:

```
Complex h1, h2, h3;
++h1;
```

Function **operator++()** returns a reference so that the object can be used as an *lvalue*

```
++h1 = h2 + ++h3;
```

How does a compiler know whether it is a pre-increment or a post-increment?

A post-fix unary operator is implemented using:
Member function with 1 dummy int argument
OR
Non-member function with two arguments

In post increment, current value of the object is stored in a temporary variable
Current object is incremented
Value of the temporary variable is returned

21.2.Post-increment operator:

```
class Complex{
...
        Complex operator ++ (int);
// friend Complex operator ++(const Complex &, int);
};

Complex Complex::operator ++ (int){
        complex t = *this;
```

```
        real += 1;
        return t;
}

Complex operator ++ (const
                        Complex & h, int){
        complex t = h;
        h.real += 1;
        return t;
}
```

How does a compiler know whether it is a pre-increment or a post-increment?

The dummy parameter in the operator function tells compiler that it is post-increment

Example:

```
Complex h1, h2, h3;
h1++;
h3++ = h2 + h3++; // Error…
```

The *pre* and *post* decrement operator **--** is implemented in exactly the same way

21.3.Type Conversion

The compiler automatically performs a type coercion of compatible types

e.g:

```
int f = 0.021;
double g = 34;
```

// type **float** *is automatically converted into* **int**. *Compiler only issues a warning…*

Type Conversion

The user can also explicitly convert between types:

```
int g = (int)0.0210;
double h = double(35);
```

// type **float** *is explicitly converted (casted) into* **int**. *Not even a warning*
// is issued now…

Type Conversion

For user defined classes, there are two types of conversions
From any other type to current type
From current type to any other type

We can do the following type conversion by our self,
Conversion from any other type to current type:
Requires a constructor with a single parameter
For example,

```
String str = 135;
```

Conversion from current type to any other type:
Requires an overloaded operator

 © Virtual University of Pakistan

For example,
String str;
char * ptr = str;

Conversion from other type to current type (int to String):

```
class String{
...
public:
        String(int a);
        char * GetStringPtr()const;
};
String::String(int a){
        cout << "String(int) called..." << endl;
        char array[15];
        itoa(a, array, 10);
        size = strlen(array);
        bufferPtr = new char [size + 1];
        strcpy(bufferPtr, array);
}
char * String::GetStringPtr() const{
        return bufferPtr;
}
int main(){
        String s = 345;
        cout << s.GetStringPtr() << endl;
        return 0;
}
```

Output:

```
String(int) called…
345
```

Automatic conversion like shown above using constructor has drawbacks conversion takes place transparently even if the user didn't wanted the conversion for example see the code below, in it user can write the following code to initialize the string with a single character:

```
int main(){

        String s = 'A';
// ASCII value of A that is 65 is being taken automatically it may be what the user //
wants, perhaps user intent was String s = "A" to store A in the string object
// but instead 65 is being stored

        cout    << s.GetStringPtr()<< endl << s.GetSize() << endl;
        return 0;
```

}

Output:

```
String(int) called…
65
2
```

Keyword **explicit**

There is a mechanism in C++ to restrict automatic conversions like this using constructor code it is to use keyword **explicit** with such constructors if we have used this keyword then casting must be explicitly performed by the user.
Keyword **explicit** only works with constructors.

Example:

```
class String{
        …
public:
        …
        explicit String(int);
};

int main(){
        String s;
        s = 'A';          // Error…
        return 0;
}

int main(){
        String s1, s2;
        s1 = String(101);       // valid, explicit casting…
        // OR
        s2 = (String)204;
        return 0;
}
```

Type Conversion

There is another method for type conversion:
"Operator overloading"
It is used for converting from current type (user defined) to any other basic type or user defined type.

General Syntax:

$TYPE_1$::operator $TYPE_2$();
Like,

```
String::operator char * ();  // TYPE1 = String , TYPE2 = char *
```

 © Virtual University of Pakistan

It will convert our string object to char *.

We write such functions as member function of the class.
NO return type and arguments are specified
Return type is implicitly taken to be TYPE2 by compiler

Type Conversion

Overloading pre-defined types:

```
class String{
        …
public:
        …
        operator int();
        operator char *();
};
String::operator int(){
        if(size > 0)
                return atoi(bufferPtr);
        else
                return -1;
}

String::operator char *(){
        return bufferPtr;
}


int main(){
        String s("2324");
        cout << (int)s << endl << (char *)s;
  //    int a = (int)s;
  //    int a = int (s);
        return 0;
}
```

Output:

2324
2324

21.4.User Defined types:

User-defined types can be overloaded in exactly the same way
Only prototype is shown below:

```
class String{
        …
        operator Complex();
        operator HugeInt();
        operator IntVector();
```

```
};
```

21.5. Drawbacks of Type Conversion Operator:

```
class String{
        …
public:
        …
        String(char *);
        operator int();
};

int main(){
        String s("Fakhir");
        // << is NOT overloaded
        cout << s; // compiler is automatically converting s to int
        return 0;
}
```

Output:

Junk Returned…

To avoid this problem DO NOT use *type conversion operators* instead use separate member function for such type conversion as shown below,

Modifying String class:

```
class String{
        …
public:
        …
        String(char *);
        int AsInt();
};
int String::AsInt(){
        if(size > 0)
                return atoi(bufferPtr);
        else
                return -1;
}
int main(){
        String s("434");
        // << is NOT overloaded
        cout << s; //error
        cout << s.AsInt();
        return 0;
}
```

 © Virtual University of Pakistan

Lecture No.22

22.1.Practical implementation of Inheritance in c++

Topics to be discussed in this lecture are,

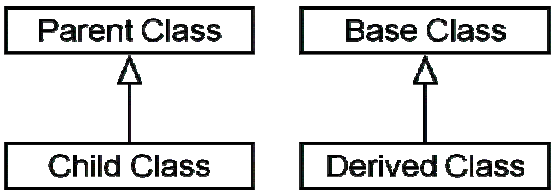
- Inheritance in Classes
- UML Notation of Inheritance
- Types of Inheritance in c++
- IS A relationship
- Accessing members
- Allocation in memory
- Constructors
- Base class initializers
- Initializing members
- Destructors
- Order of execution constructors and destructors
- Examples of C++ implementation of inheritance

22.2.Inheritance in Classes

If a class B inherits from class A, then B contains all the characteristics (information structure and behavior) of class A.
The class whose behavior is being inherited is called base class and the class who inherits the behavior of base class is called derived class. Base class is also called parent class and child class is called also derived class
Besides inherited characteristics, derived class may have its own unique characteristics

22.3.UML Notation

We use arrow from derived class to the parent class to show inheritance as shown below,



22.4.Inheritance in C++

In c++ we can inherit a class from another class in three ways,

- Public
- Private
- Protected

22.5. "IS A" Relationship

Inheritance represents "IS A" relationship for example "a student IS A person".
In general words we can say that inheritance represents,

"Derived class IS A kind of Parent class"

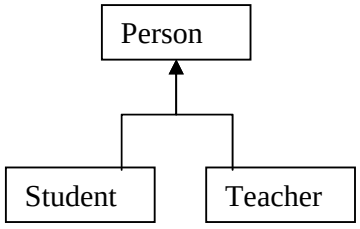
C++ Syntax of Inheritance

```
class ChildClass
        : public BaseClass{
...
};
```

Example

```
class Person{
    ...
};
class Student: public Person{
...
};
```

Accessing Members
Public members of base class become public member of derived class.
Private members of base class are not accessible from outside of base class, even in the derived class (Information Hiding)
Example
In the code given below Student and Teacher classes has been derived from single Person class,




```
class Person{
  char *name;
  int age;
  ...
public:
  const char *GetName() const;
  int GetAge() const;
  ...
};
```

 © Virtual University of Pakistan

```
class Teacher: public
   Person{
   char * dept;
   int course;
   ...
public:
   char * GetDept() const;
   int GetCourse() const;
   void Print() const;
   ...
};
```

```
class Student: public
   Person{
   int semester;
   int rollNo;
   ...
public:
   int GetSemester() const;
   int GetRollNo() const;
   void Print() const;
   ...
};
```

Example

```
void Student::Print()
                    Error
{
        cout << name << " is in" << " semester " << semester;
}
```

corrected Code:

```
void Student::Print()
{
        cout << GetName() << " is in semester " << semester;
}
int main(){
        Student stdt;

        stdt.semester = 0;//error
        stdt.name = NULL; //error
        cout << stdt.GetSemester();
        cout << stdt.GetName();

        return 0;
}
```

Explanation of above code (char * data type)

In C++ char arrays (char []) are handled in two ways one way is statically using statements like,

```
char name[30]; // static array of length 30 characters
```

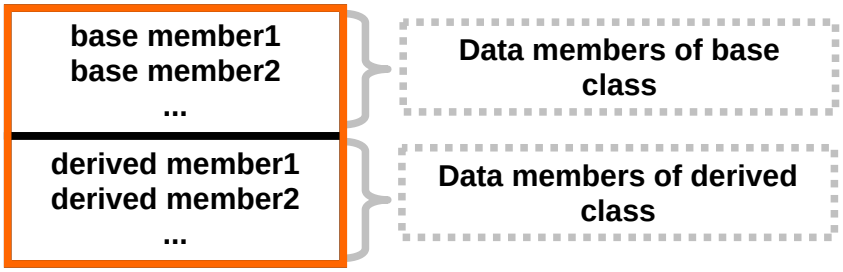
or dynamically as shown below,

```
char * name;
name = new char[30];
```

In dynamic creation of arrays we simply store char * in class and assign it a dynamic memory according to our need using new operator.

Allocation in Memory

The object of derived class is represented in memory as follows

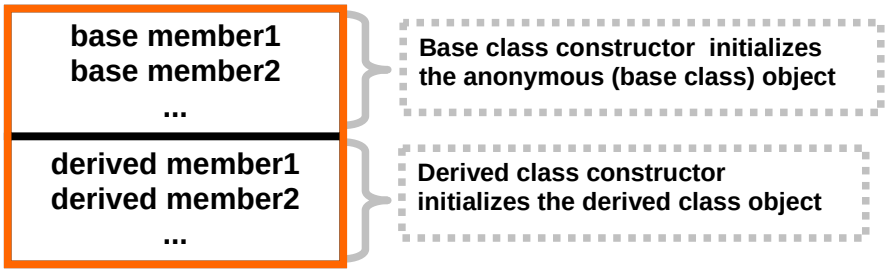


Derived Class Object

Every object of derived class has an anonymous object of base class

Constructors

- The anonymous object of base class must be initialized using constructor of base class
- When a derived class object is created the constructor of base class is executed before the constructor of derived class



Derived Class Object

Example

```
class Parent{
public:
        Parent(){ cout <<
                "Parent Constructor...";}
};
class Child : public Parent{
public:
        Child(){        cout <<
                "Child Constructor...";}
};
```


© Virtual University of Pakistan

```
int main(){
        Child cobj;
        return 0;
}
```

Output:

```
Parent Constructor...
Child Constructor...
```

Constructor

- If default constructor of base class does not exist then the compiler will try to generate a default constructor for base class and execute it before executing constructor of derived class
- If the user has given only an overloaded constructor for base class, the compiler will not generate default constructor for base class

Example

```
class Parent{
public:
        Parent(int i){}
};
class Child : public Parent{
public:
        Child(){}
} Child_Object; //ERROR
```

Definition of Some Terms:

Default constructor: Default constructor is such constructor which either has no parameter or if it has some parameters these have default values. The benefit of default constructor is that it can be used to create class object without passing any argument.

Implicit Default constructor:

Compiler generates implicit default constructor for any class in case we have not given any constructor for the class.

Explicit Default constructor:

If user has given constructor for any class without any arguments or with all arguments with default values then it is also default constructor according to definition but it is explicit (user defined) default constructor.

Now if a base class has only **non-default constructor** (constructor with parameters without default values), then when we will create object of any class derived from this base class compiler will not be able to call base class constructor as base class has no default constructor (constructor that can be called without giving any parameters) so compiler will generate error.

We can avoid this error by calling base class non-default constructor in derived class constructor initializer list by ourself.

Base Class Initializer

- C++ has provided a mechanism to explicitly call a constructor of base class from derived class

- The syntax is similar to member initializer and is referred as base-class initialization

Example

```
class Parent{
public:
        Parent(int i){…};
};
class Child : public Parent{
public:
        Child(int i): Parent(i)
        {…}
};
```

Example

```
class Parent{
public:
        Parent(){cout <<
                "Parent Constructor...";}
        ...
};
class Child : public Parent{
public:
        Child():Parent()
        {cout << "Child Constructor...";}
        ...
};
```

Base Class Initializer

- User can provide base class initializer and member initializer simultaneously

Example

```
class Parent{
public:
        Parent(){…}
};
class Child : public Parent{
        int member;
public:
        Child():member(0), Parent()
        {…}
};
```

Base Class Initializer

- The base class initializer can be written after member initializer for derived class

© Virtual University of Pakistan

- The base class constructor is executed before the initialization of data members of derived class.

Initializing Members

- Derived class can only initialize members of base class using overloaded constructors
  - Derived class can not initialize the public data member of base class using member initialization list

Example

```
class Person{
public:
        int age;
        char *name;
        ...
public:
        Person();
};
```

Example

```
class Student: public Person{
private:
        int semester;
...
public:
        Student(int a):age(a)
        {                                       //error
        }
};
```

Reason

- It will be an assignment not an initialization

Destructors

- Destructors are called in reverse order of constructor called
- Derived class destructor is called before the base class destructor is called

Example

```
class Parent{
public:
        Parent(){cout <<"Parent Constructor";}
        ~Parent(){cout<<"Parent Destructor";}
};

class Child : public Parent{
public:
        Child(){cout << "Child Constructor";}
        ~Child(){cout << "Child Destructo";}
};
```

Example

Output: **Parent Constructor**

```
Child Constructor
Child Destructor
Parent Destructor
```


© Virtual University of Pakistan

Lecture No.23

Lecture Contents:

- **Protected Access Specifier in Inheritance**
- **Implicit and explicit use of IS A relationship**

Previous lecture discussion:

Definition of Some Terms:

Default constructor: Default constructor is such constructor which either has no parameter or if it has some parameters these have default values. The benefit of default constructor is that it can be used to create class object without passing any argument.

Implicit Default constructor:
Compiler generates implicit default constructor for any class in case we have not given any constructor for the class.

Explicit Default constructor:
If user has given constructor for any class without any arguments or with all arguments with default parameters then it is also default constructor according to definition but it is explicit default constructor.

Now if a base class has only **non-default constructor** (implicit or explicit), then when we will create object of any class derived from this base class compiler will not be able to call base class constructor as base class has no default constructor (constructor that can be called without giving any parameters) so compiler will generate error.
We can avoid this error by calling base class non-default constructor in derived class constructor initializer list.

23.1. Accessing base class member functions in derived class:

Public methods of base class can directly be accessed in its derived class (***derived class interface consists of its own member functions plus member functions of its base class***).

However there are some class members functions that are written just as helper functions for other class member functions and they need not to be called directly using class object for example,

1. Suppose some function in our class wants to get some input from user in the form of integers only for this we can write another function that checks whether the entered string by user consists of integers only.

2. Suppose our class has implement encryption of data, and it encodes and decodes data using some functions these functions will also to helper functions and should not be accessible to outside word.

3. Similarly take the example of our Date class we studied in lecture no.13 has a function to check whether an year is leap year or not and according to result it allows or disallows the days of February to be 29. This bool **IsLeapYear(int)** will used by other members functions of Date class like void setDay(int) and void addDay(int), this functions is also helper function.

These helper functions are made private as there is no need to access them using class object directly as shown below,

```
class Date{
        int day, month, year;
        static Date defaultDate;
public:
        void SetDay(int aDay);
        int GetDay() const;
        void AddDay(int x);
        …
         static void SetDefaultDate(int aDay,int aMonth, int aYear);
        ...
private:

        bool IsLeapYear(int );
};

int main(){
        Date aDate;
        aDate.IsLeapYear( year );
        return 0;
}
```

Error:
bool IsLeapYear(int) is private method of Date class so it is not accessible here.

Making such functions private works fine as long as we don't derive any child class from this class, but when we derive some class for specialization these functions will not be accessible in the derived class as private members functions are accessible only in those class to which they belong but these functions are needed in derived classes as well example is given below,

Example:
Suppose we specialize our Date class by adding a child class SpecialDate to handle only working days in the an year, we will need to use **IsLeapYear** function in this child class as well but **IsLeapYear** function is private member function of base Date class (in accordance with the principles of data hiding and encapsulation), so it will not be accessible in child class as shown below,

```
class SpecialDate: public Date{
        …
public:
        void AddSpecialYear(int i){
                ...
                if(day == 29 && month == 2
                &&      !IsLeapyear(year+i)){       //ERROR!
                        ...
```

bool IsLeapYear(int) is private method of Date class so it is not accessible in its derived SpecialDate class.

© Virtual University of Pakistan

```
            }
        }
};
```

Solution:

Modify Access Specifier
One solution of this problem is that we make "IsLeapYear" function public from private as shown below,

```
class Date{
public:
        ...
        bool IsLeapYear(int );
};
void SpecialDate :: AddSpecialYear (int i) {
        ...
        if(day == 29 && month == 2
                &&     !IsLeapyear(year+i)){
                ...
        }
}
```

But the problem with making this function public is that now this functions will be accessible to everyone using Date or SpecialDate class object but we don't want this, we want that our base or derived class can access this function only. the solution of this problem is protected access specifier.

23.2."*protected*" access specifier:

c++ provides us with ***protected access specifier*** for these sorts of situations; protected access specifier ensures that function in base class is accessible in derived class of this base class and NOT outside of this class. So we can say that scope of protected access specifier is somewhere in between private and public access specifiers it is similar to private in the sense that it doesn't allow the external world to access protected member and it is similar to public access specifier in the sense that it ensures that protected member is accessible in derived classes of the protected member class.

Protected members of a class can not be accessed outside the class but only in the derived class of that class. Protected members of base class become protected member of derived class.[10]

```
class Date{
        …
```

[10] This is the case in public and protected inheritance whereas in private inheritance protected members become private members of derived class public, protected and private inheritance has been discussed in detail in lecture no.26.

```
protected:
                bool IsLeapYear(int );
};

int main(){
        Date aDate;
        aDate.IsLeapYear(); //Error
        return 0;
}
void SpecialDate :: AddSpecialYear (int i) {
        ...
        if(day == 29 && month == 2
                &&      !IsLeapyear(year+i)){
                ...
        }
}
```

Now it is ok to call function **bool IsLeapYear** in derived class *SpecialDate*.

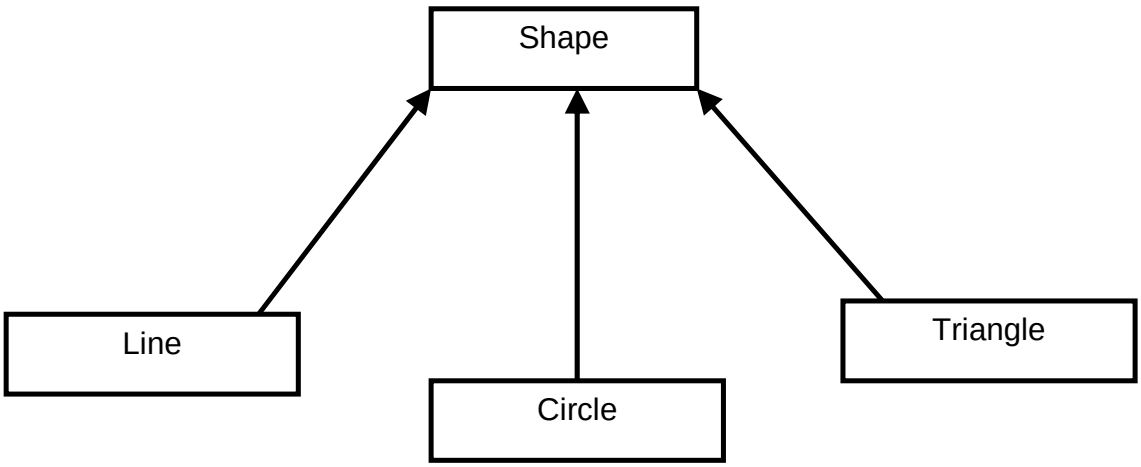
Disadvantages of protected Members:

Breaks encapsulation

The protected member is part of base class's implementation as well as derived class's implementation. So we can say that protected members breaks the principle of Encapsulation to some extent which says "A class data members and functions should be encapsulated in the class itself"

23.3. "IS A" Relationship

We have seen previously that Public inheritance models the "IS A" relationship for example see the diagram below,



Here,

- Line IS A Shape

 © Virtual University of Pakistan

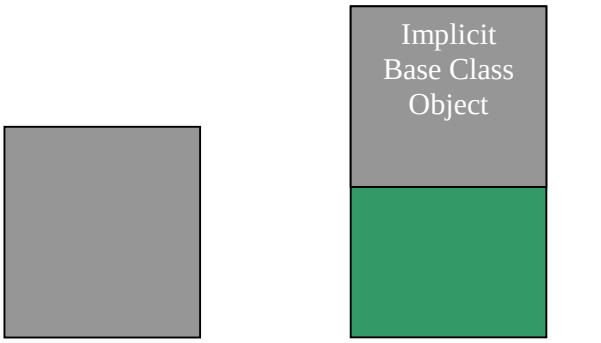
- Circle IS A Shape
- Triangle IS A Shape

Generally we can say that,

"Derived Object IS A kind of Base Object"

It means that derived class object is special kind of base object with extra properties (attributes and behaviour) so derived object has its own properties as well as properties of its base object this is the reason why base class constructor is also called while creating derived class object.

We can use derived class object where base class object is required as **derived class object has implicit base class object** also but the reverse of this statement is not true, we can not use base class object where derived class object is required because if we create base class it will only have base part not its derived class part.



Base Class Object Derived Class Object

Example

See the code below in it first we are using derived class object sobj to call methods *GetName* and *GetRollNo* belonging to classes Person and Student respectively, this is in accordance to the principle of inheritance that says child class object can access public and protected members of base class.

```
class Person {
        char * name;
public: ...
        const char * GetName();
};
class Student: public Person{
        int rollNo;
public: ...
```

```
        int GetRollNo();
};

int main()
{
        Student sobj;
        cout << sobj.GetName();
        cout << sobj.GetRollNo();
        return 0;
}
```

Now as we discussed previously that inheritance represents IS A relationship meaning relationship like,

Derived class "IS A" Base class

Hence Derived Class Object can be used where its Base Class object is required but Base Class Object can not be used where Derived Class Object is required example given below explain this point,

```
int main(){
        Person * pPtr = 0; // base class pointer
        Student s; // derived class object
        pPtr = &s;
/* assigning derived class object address to base class pointer derived class pointer is converted to base class pointer */
        cout << pPtr->GetName(); // calling base class methods using base class reference
        return 0;
}
```

So parent class object reference can hold the reference of derived class object as shown below,

```
pPtr = &s;
```

But parent class reference will have access only to the interface of parent class, if we try to access derived class functions from this parent class reference error will be generated as shown below,

```
int main(){
        Person * pPtr = 0;
        Student s;
        pPtr = &s;
        cout << pPtr->GetRollNo();              //Error
        return 0;
}
```

Compiler use the concept of static typing to decide the access to member functions as described below,

 © Virtual University of Pakistan

23.4.Static Type

The type that is used to declare a reference or pointer is called its static type

1. In Person * pPtr = 0;
   The static type of **pPtr** is Person * (Person pointer).
2. Student s;
   The static type of **s** is Student.

Member Access

As the static type of pPtr is Person so following call is erroneous

pPtr->GetRollNo();

As pPtr static type is Person pointer and in Person class there is no GetRollNo() function so compiler will generate an error.

In c++ we can also use references (called as reference identifiers or reference variables or reference constants) to any type instead of pointers, for example see the example below where the reference of base object is being initialized with derived class object,

Example (Explicit use of IS A relationship)

```
int main(){

Person p;
Student s;
Person & refp = s;

/*Here refp is becoming reference (alias) of Student object but as its own static type is Person so it can access member functions of Person class only. */

cout << refp.GetName();
cout << refp.GetRollNo(); //Error
return 0;
}
```

Example (Implicit use of IS A relationship)

```
  Play(const Person& p){
        cout << p.GetName()
                  << " is playing";
}
void Study(const Student& s){
        cout << s.GetRollNo()
                   << " is Studying";
}

int main(){
Person p;
```

```
Student s;
Play(p);/* parameter of function Play is being initialized with argument p */
Play(s);
/* parameter of function Play is being is initialized with argument s as student IS
A kind of person*/
return 0;
}
```


© Virtual University of Pakistan

Lecture No.24

Lecture Overview:

Protected Access Specifier
IS-A relationship
Copy constructor and inheritance

Protected Members:

Protected members are somewhere between public and private members. They are used in inheritance.
From outside the class no once can access protected access specifier however any publicly derived class can access the protected access specifiers and they behave as protected members of derived class.
In a single standalone class protected access specifier has no significance but when we derive any class from this class protected access specifier becomes significant.

IS-A relationship:

Inheritance is used when two classes have IS A kind of relationship between two classes.
In c++ public inheritance is used for IS A relationship.
We can use publicly derived class pointer for its base class pointer because derived class is a kind of base class or in other words derived class is base class with some extra properties. In case of "Student IS-A Person relationship" student has all properties of person for example student walks, eats, drinks and so on as person does but student has some extra properties like student studies is some study program.

(We can assign derived class pointer to its base class pointer because for compiler it is explicit manifestation of IS-A relationship.)

Static type of an identifier:

Static type of an identifier is type used to declare it.
Static type of an identifier is the type used for declaration of that identifier for example,

```
Student * student;
Person * person;
```

Here *static type* of pointer *student* is Student and *person* is Person.

Access to members of an identifier is governed by static type of that identifier, compiler uses static type for an identifier to allow the identifier to use different members (functions and variables).

Copy Constructor:[11]

Copy constructor is a member function of a class that is used to create an object of that class ***by copying values from an already created object.***

Copy Constructor in case of Inheritance:

Consider the code given below having two classes Person and Student in this code Person is base and Student is derived class. Person has one attribute ***name*** to store name of the person and Student has one attribute ***major*** to store study program of the student,

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



/*Listing 24.1 */
/*Program to demonstrate Copy Constructor in case of base and derived classes. */

/*    Base Class Person    */
class Person{
        char * name;
public:
        Person(char * = NULL);
        const char * GetName() const;
        ~Person();
};

Person::Person(char * nm): name(NULL){

   if (nm != NULL)
      {
        name = new char[strlen(nm)+1];
        strcpy(name,nm);
      }

   }
const char * Person::GetName() const {
   if (name != NULL )
   return name;
   else
```

[11] See Lecture No.9 for more details about copy constructor.

 © Virtual University of Pakistan

```
   return "NULL";

   }

Person::~Person(){
delete []name;
   }

/*                  Child Class Student                                    */
class Student: public Person{
          char* major;
public:
          Student(char * , char * );
          void Print() const;
          ~Student();
};

Student::Student(char *_name, char *_maj) : Person(_name), major(NULL)
{
   if (_maj != NULL) {
          major = new char [strlen(_maj)+1];
          strcpy(major,_maj);
   }
}
void Student::Print() const{
    cout << "Name: "<< GetName() << endl;
    cout << "Major: " << major  << endl;

}

Student::~Student(){
          delete []major;
}

int main(){

          Student sobj1("Ali","Computer Science");
          Student sobj2 = sobj1;
             //Student sobj2(sobj1);   Similar to above statement to call copy constructor
   sobj2.Print();
          system("PAUSE");
          return 0;

}
```

Code Description:

The code has two classes Person and Student this code is also manifestation of IS-A kind of relationship (public inheritance in c++).

Person class has an attribute name to store name of any person as all persons have specific names.
Student class has an attribute major to store the major of the student.

Person has the following in its public interface:

```
Constructor          // one parameter name with default value NULL
GetName()            // getter function
Destructor()         // used due to dynamic memory allocation
```

Student has the following in its public interface:

```
Constructor     // two parameter constructor with default values NULL
Print()         // showing student attributes
Destructor      // used to free memory dynamically allocated
```

The constructor's code of student class is as shown below,

```
Student::Student(char *_name, char *_maj) : Person(_name), major(NULL)
{
   if (_maj != NULL) {
         major = new char [strlen(_maj)+1];
         strcpy(major,_maj);
   }
}
```

The constructor of student class is taking two char *'s as inputs, one is name of student and second is major subject of that student.
In initialization list we are initializing major with null value and we are also calling base class constructor of person class as we discussed today base class part of an objects is created first and then its derived class part is created.
Then there is code of student constructor in which are checking maj for NULL if it is not (some value has passed for maj subject) we are creating memory equal to passed value and assigning it to char * major.

The print function code of student class is given below,

```
void Student::Print() const {
    cout << "Name: "<< GetName() << endl;
    cout << "Major: " << major  << endl;

}
```

In print method we are doing two things first we are printing student name using person GetName method and then we are showing student major.

© Virtual University of Pakistan

The main function is given below,

```
int main(){

        Student sobj1("Ali", "Computer Science");
        Student sobj2 = sobj1;
            //Student sobj2(sobj1);     Similar to above statement to call copy constructor
       sobj2.Print();
        system("PAUSE");
        return 0;

}
```

In main we are creating one object of Student class with name ***sobj1*** with values "Ali" and "Study Program" for name and major,

Student sobj1("Ali","Computer Science");

Then we are assigning ***sobj1*** to a new student object ***sobj2*** with the line given below,

Student sobj2 = sobj1;

This line will invoke copy constructor of Student class as we are creating an object of student class in terms of another object that already exist for student class.

This line is exactly same as commented line,

//Student sobj2(sobj1);

Here one object **sobj1** values have been assigned to newly created object using **default compiler generated copy constructo**r, this copy constructor is created by compiler in the same way as compiler generates default constructor in case user has not defined default constructor for that class.

Then we are calling print method of student class to ensure that we values have been assigned to new object as well,

sobj2.Print();

The output of this code will be as follows:

C++ Out Put:

```
Name: Ali
Major: Computer Science
```

As we know compiler may generate copy constructor for any class, if needed. In this case when we are assigning derived class object **sobj1** to another newly created

object **sobj2**, derived class object Copy constructor will be invoked which in turn will call the copy constructor of the Base class as base class anonymous object is created first and then derived part is created.

Shallow Copy

Compiler by default uses shallow copy so compiler generated **default copy constructor** will also use shallow copy which simply copies all values of already existing object to newly created object as shown in the diagram below,

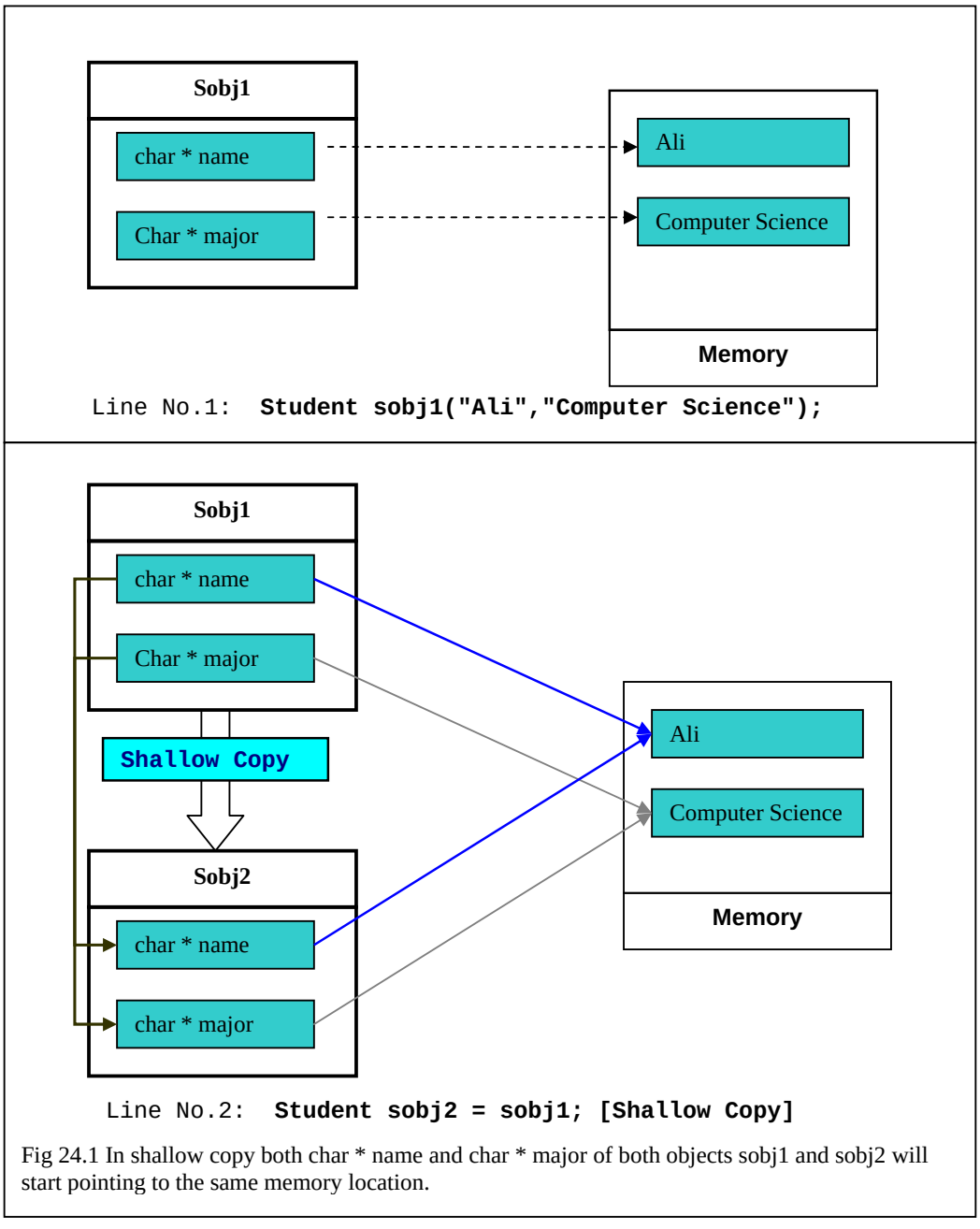


Fig 24.1 In shallow copy both char * name and char * major of both objects sobj1 and sobj2 will start pointing to the same memory location.

 © Virtual University of Pakistan

Compiler generates copy constructor for derived class, calls the copy constructor of the base class and then performs the shallow copy of the derived class's data members.

The problem with shallow copy remains same as was in the case of copy constructor for a single class that it doesn't work correctly in case of dynamic memory because in that case we are only storing pointer of the dynamic memory in class when we will do copy only the pointer value will be copied and pointers of both objects of student class will start pointing to same memory location as shown in the diagram now suppose obj1 is destroyed it frees it's memory allocated using new operator using its destructor the result will be that obj2 pointer will also become invalid. The solution is Deep Copy in deep copy we will write copy constructor code by our self to ensure that when one object is copied from other object new dynamic memory is allocated for it as well so that it doesn't rely on previous object memory.

Deep Copy:

Let we first write code for deep copy in our base class Person.
For this we will write our own copy constructor as we wrote our own default constructor as shown below,

```
Person::Person(const Person & rhs): name(NULL){
   /* Code for deep copy*/
   if (rhs.name != NULL)
      {
         name = new char[strlen(rhs.name)+1];
         strcpy(name,rhs.name);
      }

   }


int main(){

         Student sobj1("Ali","Computer Science");
         Student sobj2 = sobj1;
            //Student sobj2(sobj1);  Similar to above statement to call copy constructor
   sobj2.Print();
         system("PAUSE");
         return 0;

}
```

When we will run this code the output will again be same,

C++ Out Put:

```
Name: Ali
Major: Computer Science
```

In this case as we have written our self written copy constructor of base class so Compiler will generate copy constructor for only derived class, will call our written copy constructor of the base class so it will perform the **Shallow copy** for derived class's data members and **Deep copy** for base class as our base class has our own written copy constructor with **Deep copy** code.

Now let us write our own copy constructor for derived class as well as shown below,

```
Student::Student(const Student & rhs) : major(NULL)
{
   if (rhs.major != NULL) {
         major = new char [strlen(rhs.major)+1];
         strcpy(major,rhs.major);
   }
}

int main(){

         Student sobj1("Ali","Computer Science");
         Student sobj2 = sobj1;
            //Student sobj2(sobj1);   Similar to above statement to call copy constructor
   sobj2.Print();
         system("PAUSE");
         return 0;

}
```

But now when we executed this code the output is as follows:

C++ Out Put:

```
Name: NULL
Major: Computer Science
```

Name of sobj2 was not copied from sobj1 or we can say that only copy constructor of derived class executed but base class copy constructor did not execute.
The reason for this error is that we have not explicitly called copy constructor of base class from derived class. And compiler called base class default constructor by itself to complete the object creation process.

© Virtual University of Pakistan

We can check whether base class default constructor was called or not by adding a cout statement in our base class default constructor as shown below,

24.1.Modified Default Constructor

```
Person::Person(char * nm): name(NULL){

  if (nm != NULL)
    {
      name = new char[strlen(nm)+1];
      strcpy(name,nm);
    }
  else {
       cout<<"In Person default constructor..\n";
    }


  }

int main(){

      Student sobj1("Ali","Computer Science");
      Student sobj2 = sobj1;
        //Student sobj2(sobj1);   Similar to above statement to call copy constructor
      sobj2.Print();
      system("PAUSE");
      return 0;

}
```

Now when we executed this code the output was,

C++ Out Put:

```
In Person default constructor..
Name: NULL
Major: Computer Science
```

Now we modify our derived class copy constructor code such that it explicitly call copy constructor of base class from its initialization list,

Example

```
Person::Person(const Person & rhs): name(NULL){

  if (rhs.name != NULL)
    {
      name = new char[strlen(rhs.name)+1];
```

```
        strcpy(name,rhs.name);
    }

}

Student::Student(const Student & rhs) : Person(rhs), major(NULL)
{
   if (rhs.major != NULL) {
        major = new char [strlen(rhs.major)+1];
        strcpy(major,rhs.major);
   }
}
```

Now our main function shown previously will give following output

C++ Out Put:

```
Name: Ali
Major: Computer Science
```

The output again is correct now so this is our final code with user defined copy constructors for both base and derived class and derived class copy constructor *calling* base class copy constructor.

In table below we have shown the steps number wise involved in creation of an object using copy constructor when we executed following lines in main,

```
int main(){

        Student sobj1("Ali","Computer Science");
        Student sobj2 = sobj1;
// Here copy constructor of student class will be called
        return 0;

}
```

3	Person::Person(const Person &rhs) :
4	name(NULL) {
5	//code for deep copy
	}
1	Student::Student(const Student & rhs) :
6	major(NULL),
2	Person(rhs){
7	//code for deep copy
	}

Table 24.1

 © Virtual University of Pakistan

Assignment Operator

Compiler also generates code for *assignment operator* for a class, if needed. In the case of inheritance when we assign one derive class object to another derive class object compiler uses assignment operator. Derived class copy assignment operator is invoked *which in turn calls* the assignment operator of the base class.
There are lot of similarities between copy constructor and assignment operator these are,
In case our class involves dynamic memory allocation we had to write assignment operator code by our self as we had to write the user defined code for copy constructor.

Similarly derived class assignment operator has to call base class assignment operator to complete the assignment process as derived class also contains implicit base class part in case of compiler generated assignment operator compiler does it by itself but if case we programmer has to call assignment operator of base class explicitly in case of user defined assignment operator.

The example given below explain this concept in detail,

Example

```
class Person{
public:
        Person & operator =
                                (const Person & rhs){
                cout << "Person Assignment";
                // Code for deep copy assignment
        }
};
Example
class Student: Public Person{
public:
        Student & operator = (const Student & rhs){
                cout<< "Student Assignment";
                // Code for deep copy assignment
        }
};
int main()
{
        Student sobj1, sboj2("Ali", "CS");
        sobj1 = sobj2;
        return 0;
}
```

Output

Student Assignment

The assignment operator of base class is not called

Calling Base Class member functions:
We can not call base class assignment operator from derived class using initialization list.

There are two ways of calling base class function from derived class one is calling base class function explicitly (mentioning base class name) and the other is calling base class function implicitly (without mentioning base class name).

Explicit Way:

In the code below we are calling base class **const char *GetName()** method explicitly,

```
const char * Person::GetName() const {

   ...
   ...

}


void Student::Print() const{

    //cout << "Name: "<< GetName() << endl;
    cout << "Name: "<< Person::GetName() << endl;

/*explicit call to base class GetName method (if we even do not write base class name even then call will work as derived class can call its base class public methods*/

    cout << "Major: " << major  << endl;

}
```

Implicit Way:

We can call the same method implicitly as well as shown below,

```
const char * Person::GetName() const {

   ...
   ...

}
void Student::Print() const {
```

 © Virtual University of Pakistan

```
 /* Implicit call to base class GetName method using this pointer*/
 cout <<"Name: "<<static_cast<const Person &>(*this).GetName()<<endl;
 cout << "Major: " << major  << endl;

}
```

Assignment Operator

In the same way we can also call assignment operator in base class in two ways also,
Calling assignment operator of base class explicitly
Calling assignment operator of base class implicitly

Explicitly Calling operator =

```
Student & Student ::operator =(Student & rhs){


   Person::operator = (rhs);  /*Explicit call to base class*/
   /*Student object rhs will be converted to Person object in function call
   above as derived class IS A is a kind of Base class*/

   /*Implicit calls to base class part*/
   //static_cast<Person &>(*this) = rhs;
   //Person(*this) = rhs;
   //(Person)*this = rhs;
   /*All the above three statements have the same meaning.*/

   if (major != NULL)
   delete [] major; // deleting previous allocated memory for major

   if(rhs.major != NULL){
   major = new char[strlen(rhs.major)+1];
   strcpy(major,rhs.major);
   }

}
```

Implicitly Calling operator =

```
Student & Student ::operator =(Student & rhs){


   //Person::operator = (rhs);  /*Explicit call to base class*/
   /*Student object rhs will be converted to Person object in function call
   above as derived class IS A is a kind of Base class*/

   /*Implicit calls to base class part*/
   static_cast<Person &>(*this) = rhs;  /* c++ way of type casting */
   //Person(*this) = rhs;   /* c way of type casting */
```

```
   //(Person)*this = rhs;   /* c way of type casting */
   /*All the above three statements have the same meaning.*/

   if (major != NULL)
   delete [] major; // deleting previous allocated memory for major

   if(rhs.major != NULL){
   major = new char[strlen(rhs.major)+1];
   strcpy(major,rhs.major);
   }

}
```

Appendix:

Type conversion:

C++ is strongly typed language mean we can not use one type instead of other type we need to convert one type in other type before using it.
This conversion from one type to another type is called **casting**.

Casting:

Casting can be done in two ways ,

Implicit casting conversion :

Conversions that compiler can perform automatically using build in casting operations without casting request from programmer.

Explicit casting conversion:

Conversion in which programmer requests the compiler to convert one type to another.
There are four types of casting operators that programmer can use to convert one type ot another as given below,

The C++ draft standard includes the following four casting operators,

1. static_cast
2. const_cast
3. dynamic_cast
4. reinterpret_cast

In perspective of inheritance casting can be of two kinds,

a. Up casting (casting from derived to base class)
b. Down casting (casting from base to derived class)

 © Virtual University of Pakistan

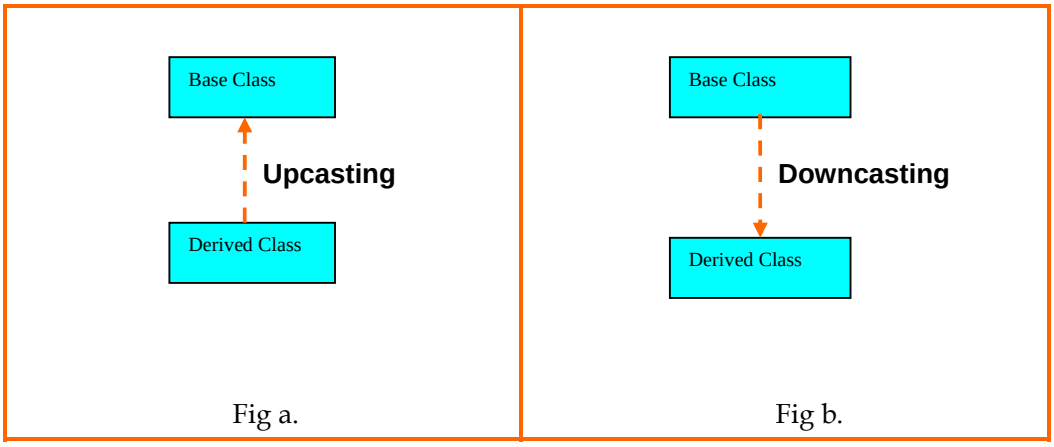


Fig a. Fig b.

Downcasting:

Example: (Base to Derive class)

Dynamic cast: (polymorphic behavior)

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

using namespace std;

class Base{
public:
virtual void function() { cout<<"I am in Base"<<endl;  }
};
class Derived : public Base {
public:
void function() { cout<<"I am in Derived"<<endl;  }
};
int main()
{
Base * base = new Derived();
base->function();
Derived * derived = dynamic_cast<Derived*>(base);
derived->function();
system("PAUSE");
return 0;
}
```

Reinterpret cast:

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

using namespace std;
```

```
class Base{
public:
void function() { cout<<"I am in Base"<<endl;  }
};
class Derived : public Base {
public:
void function() { cout<<"I am in Derived"<<endl;  }
};
int main()
{
Base * base = new Derived();
base->function();
Derived * derived = reinterpret_cast<Derived*>(base);
derived->function();
system("PAUSE");
return 0;
}
```

Note:

1. char * is c++ built in data type that can be used exactly like char arrays in fact char arrays are also treated as char * in c++ so we used char * where we needed to use char array.
2. In video lecture at duration 24 min there should only be one memory location to store values, Ali newly created object pointer will be NULL for name and will not point to any memory location.

References:

http://www.acm.org/crossroads/xrds3-1/ovp3-1.html (Casting Tutorial see this link for more details about casting)

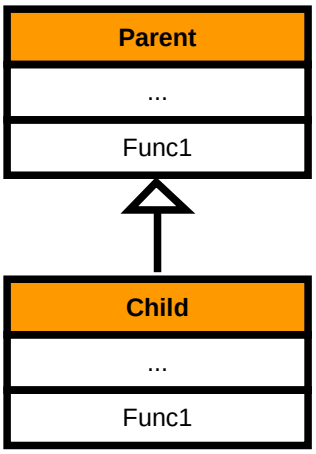
Open Source Editor Note Pad++ has been used in this document for code highlighting.

 © Virtual University of Pakistan

Lecture No.25

Overriding Member Functions of Base Class in Derived Class (Function Overriding)

Derived class can override the member functions of its base class. To override a function the derived class simply provides a function with the same signature[12] (prototype) as that of its base class

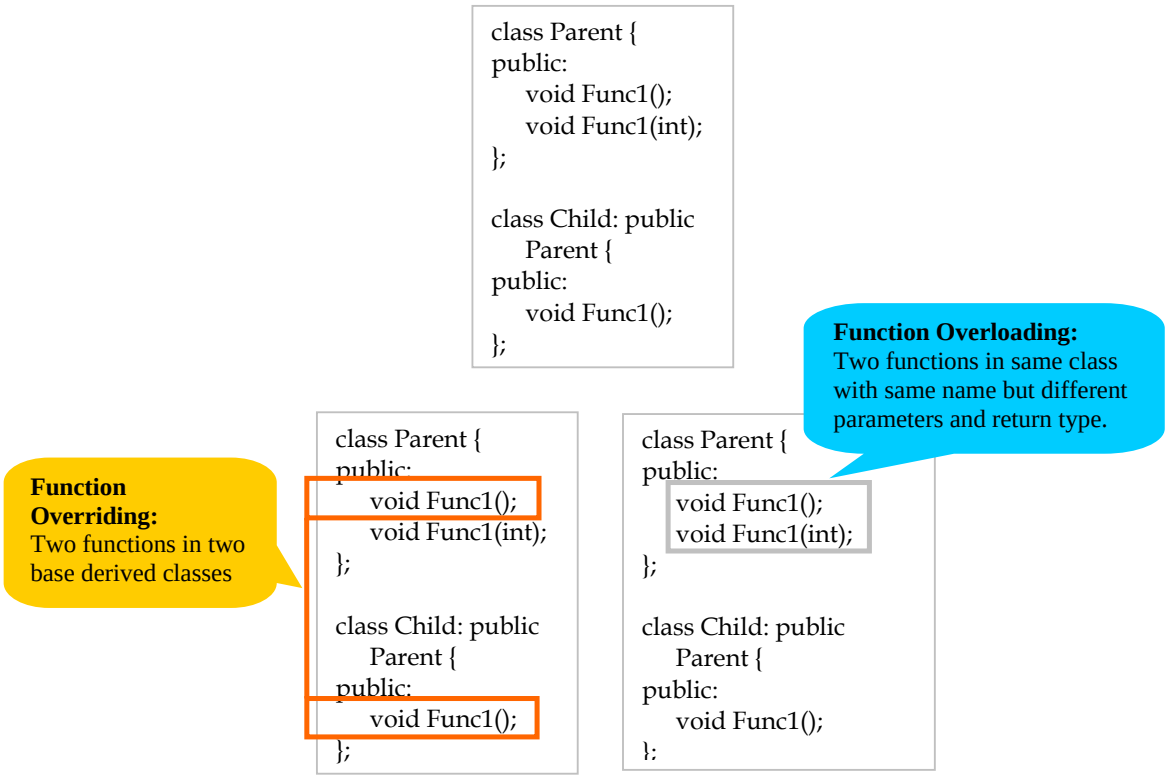


Overriding

25.1.Overloading vs. Overriding

Function Overloading is done within the scope of one class whereas Function Overriding is done in scope of parent and child classes (inheritance).

[12] same name, parameters and return type



Overriding within the scope of single class is error due to duplicate declaration (two member functions with same prototype)

```
class Parent {
public:
        void    Func1();
        void    Func1();        //Error
};
```

Overriding Member Functions of Base Class

Derive class can override member function of base class such that the working of function is totally changed.

Example

```
class Person{
public:
        void Walk();
};
class ParalyzedPerson: public Person{
public:
        void Walk();
};
```


© Virtual University of Pakistan

Overriding Member Functions of Base Class

Derive class can override member function of base class such that the working of function is similar to former implementation.

Example

```
class Person{
        char *name;
public:
        Person(char *=NULL);
        const char *GetName() const;
        void Print(){
                cout << "Name: " << name
                        << endl;
        }
};
class Student : public Person{
        char * major;
public:
        Student(char * aName, char* aMajor);
        void Print(){
                cout    <<"Name: "<< GetName()<<endl
                        << "Major:" << major<< endl;
        }
};
int main(){
        Student a("Ahmad", "Computer Science");
        a.Print();
        return 0;
}
```

Output:

```
Name: Ahmed
Major: Computer Science
```

Overriding Member Functions of Base Class

Derive class can call base class member function from its overridden member function to perform the base class part and then can perform its own tasks for example student class can call base class person method Print to show name of student and then can show study program of student by itself, this approach is in accordance with the Object Oriented Programming principles which says that each class should perform its tasks by itself, this is shown in example below,

Example

```
class Student : public Person{
        char * major;
public:
        Student(char * aName, char* m);
```

```
        void Print(){
                Print();     //Calling Print of Person to print student name
                cout<<"Major:" << major <<endl;  // Displaying study program of
student
        }
};
int main(){
        Student a("Ahmad", "Computer Science");
        a.Print();
        return 0;
}
```

But there will be no output on the screen after executing this code due to minor mistake it is due to the reason that our both classes Student and Person have methods with name Print and we are calling Print method of Person from Student Print method compiler will call to Print of Student class again and again recursively as we are calling Print method form Student class.

In this case it is necessary to mention that we are calling base class Print method as shown below,

```
class Student : public Person{
        char * major;
public:
        Student(char * aName, char* m);

        void Print(){
                Person::Print();
                cout<<"Major:" << major <<endl;
        }
...
};
```

Example

```
int main(){
        Student a("Ahmad", "Computer Science");
        a.Print();
        return 0;
}
```

Output

Name: Ahmed
Major: Computer Science

Overriding Member Functions of Base Class

 © Virtual University of Pakistan

As we see previously when we use pointers to call overridden methods they are called according to static type of pointer as shown below,

Example

```
int main(){
        Student a("Ahmad", "Computer Science");
        Student *sPtr = &a;
        sPtr->Print(); // static type of sPtr is Student * so Student Print method will be called

        Person *pPtr = sPtr; // static type of pPtr is Person * so Person Print method will be called
        pPtr->Print();
        return 0;
}
```

Output:

Name: Ahmed
Major: Computer Science

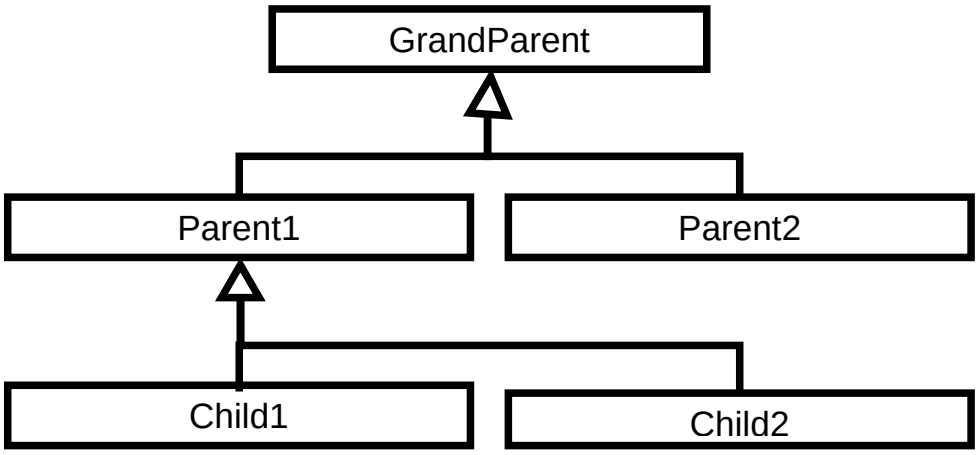
Name: Ahmed

This may be undesirable and to avoid this situation avoid using simple method overriding in this way its modified form (virtual functions will be covered in coming lectures).

25.2.Hierarchy of Inheritance

We represent the classes involved in inheritance relation in tree like hierarchy

Example



Direct Base Class

A direct base class is explicitly listed in a derived class's header with a colon (:)

```
class Child1:public Parent1 {        // Here Parent1 is Direct Base Class of Child1
…
};
```

Indirect Base Class

An indirect base class is not explicitly listed in a derived class's header with a colon (:)

It is inherited from two or more levels up the hierarchy of inheritance

```
class GrandParent {
…
};
class Parent1: public GrandParent {
…
};
class Child1:public Parent1 {        // Here GrandParent is InDirect Base Class of Child1
…
};
```

 © Virtual University of Pakistan

Lecture No.26

26.1. Base Initialization

We saw in the previous lectures, that in the case of copy constructor we have to call base class constructor from the initialization list of child class because implicit base class object is created first and then derived class object is created. Now we see this concept in detail in the perspective of class hierarchy,

- The child can only call constructor of its direct base class to perform its initialization using its constructor *initialization list.*
- The child cannot call the constructor of any of its indirect base classes to perform their initialization using its constructor *initialization list*

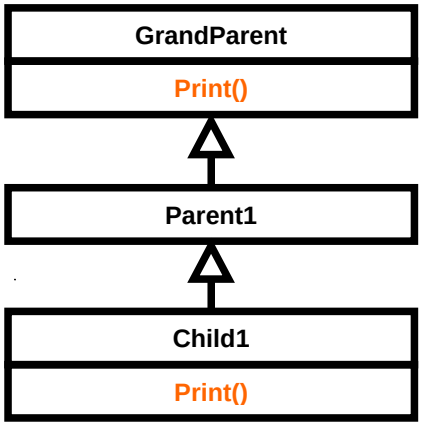
It is explained in example code given below,

Example

```
class GrandParent{
        int gpData;
public:
        GrandParent() : gpData(0){...}
        GrandParent(int i) : gpData(i){...}
        void Print() const;
};
class Parent1: public GrandParent{
        int pData;
public:
        Parent1() : GrandParent(), pData(0) {…}
};
class Child1 : public Parent1 {
public:
        Child1() : Parent1()    {...}
        Child1(int i) : GrandParent (i) //Error: Child1 can not call its indirect base class GrandParent Constructor from its constructor initialization list.
        {...}
        void Print() const;
};
```

Overriding in case of class hierarchy:

In class hierarchy Child class can override the function of any of its Parent class (direct or indirect) as shown in the diagram below (we see this concept in detail in types of inheritance given below),



Example

```
void GrandParent::Print() {
        cout   << "GrandParent::Print"
                        << endl;
}

void Child1::Print() {
        cout << "Child1::Print" << endl;
}
int main(){
        Child1 obj;
        obj.Print();
        obj.Parent1::Print();
        obj.GrandParent::Print();
        return 0;
}
```

Output

```
Child1::Print
GrandParent::Print
GrandParent::Print
```

26.2. Types of Inheritance

There are three types of inheritance

- Public
- Protected
- Private

We can use these keywords (public, private or protected) to specify the type of inheritance

 © Virtual University of Pakistan

a. Public Inheritance
class Child: public Parent {…};

Member access in	
Base Class	**Derived Class**
Public	Public
Protected	Protected
Private	Hidden

b. Protected Inheritance
class Child: protected Parent {…};

Member access in	
Base Class	**Derived Class**
Public	Protected
Protected	Protected
Private	Hidden

c. Private Inheritance
class Child: private Parent {…};

Member access in	
Base Class	**Derived Class**
Public	Private
Protected	Private
Private	Hidden

If the user does not specifies the type of inheritance then the default type is private inheritance,

```
class Child: private Parent {…}
```

is equivalent to,

```
class Child: Parent {…}
```

We have seen public inheritance in previous lecture, now we see private and protected inheritance in detail,

26.3.Private Inheritance

We use private inheritance when we want to reuse code of some class. Private Inheritance is used to model "Implemented in terms of" relationship

Example

Suppose we have a class collection to store element collection as shown below,

```
class Collection {
...
public:
        void AddElement(int);
        bool SearchElement(int);
        bool SearchElementAgain(int);
        bool DeleteElement(int);
};
```

As you can see it supports the following methods,
AddElement: to add elements in collection
SearchElement: search any element in collection it will true as soon as any element will be found
SearchElementAgain: finds second instance of any element in collection it will return true as soon as it will find any duplicate entry of any element.
DeleteElement: to delete any entry from collection

You can see that Class collection allows duplicate elements.
Suppose now we want to implement class Set, class Set has very similar functionality as that of collection class with the difference that Set class can not allow duplicate elements in it, so we can use the concept of inheritance here, we can derive class Set from class Collection.
But we can not use public inheritance here as it allows interface (all functions) of base class to be accessed using derived class object, but we don't want to allow all functions of class Collection to act on Set class object. We only want to use some functions of Collection class (Base class) in derived Set class for to implement Set class functionality so we will use private inheritance here the two main advantages we will achieve through private inheritance in this case are,

1. Specialization of class according to set class (removing extra features)
2. Making interface of collection class inaccessible from outside world using class set reference.

 © Virtual University of Pakistan

Class Set

```
class Set: private Collection {
private:
        ...
public:
        void AddMember(int);
        bool IsMember(int);
        bool DeleteMember(int);
};
void Set::AddMember(int i){
                if (! IsMember(i) )
                        AddElement(i);
}
bool Set::IsMember(int i){
        return SearchElement(i);
}
```

Lecture No.27

Previous Lecture Review:

In last lecture we saw the use of private inheritance using the example of classes **Collection and Set,** we saw that class collection was allowing duplicate elements but set class can not allow duplicate elements in it however other functionality was same, so we privately inherited class set from collection and added only those member functions in derived class **set** that were related to set class, the two main advantages we achieved through private inheritance in this case were,

1. Specialization of class collection according to set class (removing extra features)
2. Making interface of collection class inaccessible from outside world using class set reference, to avoid any unintentional use of collection class restricted features in set class.

The code of both classes in shown below,

Class Collection	Class Set

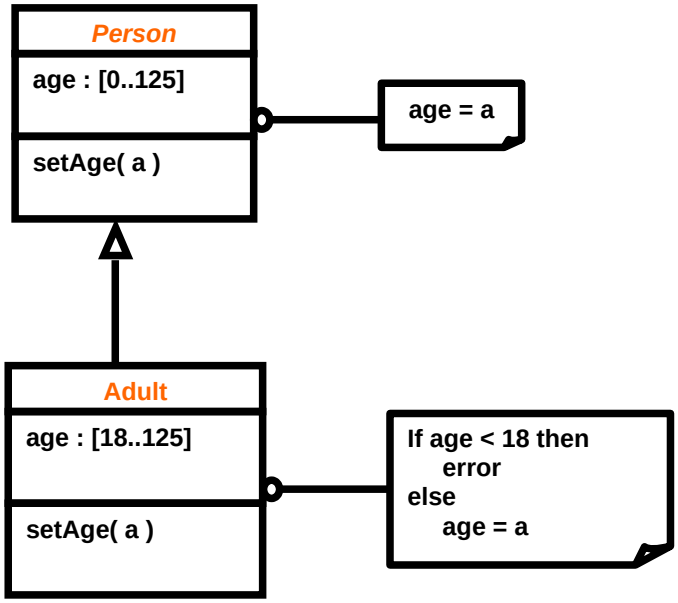
class Collection {
...
public:
void AddElement(int);
bool SearchElement(int);
bool SearchElementAgain(int);
bool DeleteElement(int);
};
	class Set: private Collection {
	private:
	...
	public:
	void AddMember(int);
	bool IsMember(int);
	bool DeleteMember(int);
	};

As we discussed that we achieve specialization using private inheritance so need to see specialization (Specialization concept was discussed in Lecture No.4) again in detail,

27.1.Specialization (Restriction)

In specialization derived class is behaviourally incompatible with the base class.
Behaviourally incompatible means that base class can't always be replaced by the derived class.
Specialization is represented by **"Implemented in terms of"** relationship.
Specialization (Restriction) can be implemented using both **private and protected inheritance.**

© Virtual University of Pakistan



Example – Specialization (Restriction)

```
class Person{
        …
protected:
        int age;
public:
        bool SetAge(int _age){
                if (_age >=0 && _age <= 125) {
                        age = _age;
                        return true;
                }
                return false;
        }
};
class Adult : private Person {
        public:
        bool SetAge(int _age){
                if (_age >=18 && _age <= 125) {
                        age = _age;
                        return true;
                }
                return false;
        }
};
```

Essential properties of Private Inheritance:

1. In Private Inheritance only member functions and friend classes or functions of a derived class can convert pointer or reference of derived object to that of parent object

Example

```
class Parent{
};
class Child : private Parent{
};

int main(){
        Child cobj;
        Parent *pptr = & cobj;      //Error
        return 0;
}
```

```
void DoSomething(const Parent &);

Child::Child(){
        Parent & pPtr = static_cast<Parent &>(*this);  // fine
        DoSomething(pPtr);
        //      DoSomething(*this); // this single line is equal to two lines above.
}
```

2. As was in the case of public inheritance child class object has an anonymous object of parent class.
3. As was in the case of public inheritance the default constructor and copy constructor of parent class are called when constructing object of derived class.

Example

```
class Parent{
public:
        Parent(){
                cout << "Parent Constructor";
        }

        Parent(const Parent & prhs){
          cout << "Parent Copy Constructor";
        }
};
class Child: private Parent{
public:
         Child(){
```

 © Virtual University of Pakistan

```
                cout << "Child Constructor";
        }

         Child(const Child & crhs)  : Parent(crhs){
                cout << "Child Copy Constructor";
        }
};
int main() {
        Child cobj1; // default constructor will be invoked
        Child cobj2 = cobj1;  // copy constructor will be invoked
        //Child cobj2(cobj1); // another way of calling copy constructor
        return 0;
}
```

Output:

```
Parent Constructor
Child Constructor
Parent Copy Constructor
Child Copy Constructor
```

4. In private inheritance the derived class that is more than one level down the hierarchy cannot access the member functions of grand parent class as public and protected members functions of derived class become private members of privately derived class for all practical purposes. For example see the code below here Child class is derived class that is more than one level down the hierarchy and hence can not access the member functions of **GrandParent** class.

```
class GrandParent{
public :
        void DoSomething();
};

class Parent: private GrandParent{
        void SomeFunction(){
                DoSomething();
        }
};
class Child: private Parent
{
public:
        Child() {
                DoSomething();                  //Error
        }
};
```

5. In private inheritance the derived class that is more than one level down the hierarchy cannot convert the pointer or reference to that of GrandParent , for example in code given below Child class can not convert its reference to its GrandParent reference. The reason is same that in private inheritance we implement specialization and we restrict all features of base class to privately derived class only and its friend classes or functions.

```
void DoSomething(GrandParent&);
class GrandParent{
};
class Parent: private GrandParent{
public:
        Parent() {DoSomething(*this);}
};
class Child: private Parent {
public:
        Child()
        {
                DoSomething(*this);          //Error
        }
};
```

27.2.Protected Inheritance

If a class D has been derived using protected inheritance from class B (If B is a protected base and D is derived class) then public and protected members of B can be accessed by member functions and friends of class D and classes derived from D.
Protected inheritance is used to build class hierarchy using **"Implemented in terms of"** relationship.

If B is a protected base and D is derived class then member functions and friends of class D and classes derived from D can access member functions of class B, (note that in private inheritance only derived class can access the member functions of base class). So we can say that protected inheritance lies between public and private inheritance.

 © Virtual University of Pakistan

```
class GrandParent{
public :
        void DoSomething();
};

class Parent: protected GrandParent{
        void SomeFunction(){
                DoSomething();
        }
};
class Child: protected Parent
{
public:
        Child()
        {
                DoSomething();
        }
};
```

27.3.Properties of Protected Inheritance

If B is a protected base and D is derived class then only friends and members of D and friends and members of class derived from D can convert D* to B* or D& to B& (in private inheritance only derived class or its friends can convert pointer to base class)

```
void DoSomething(GrandParent&);

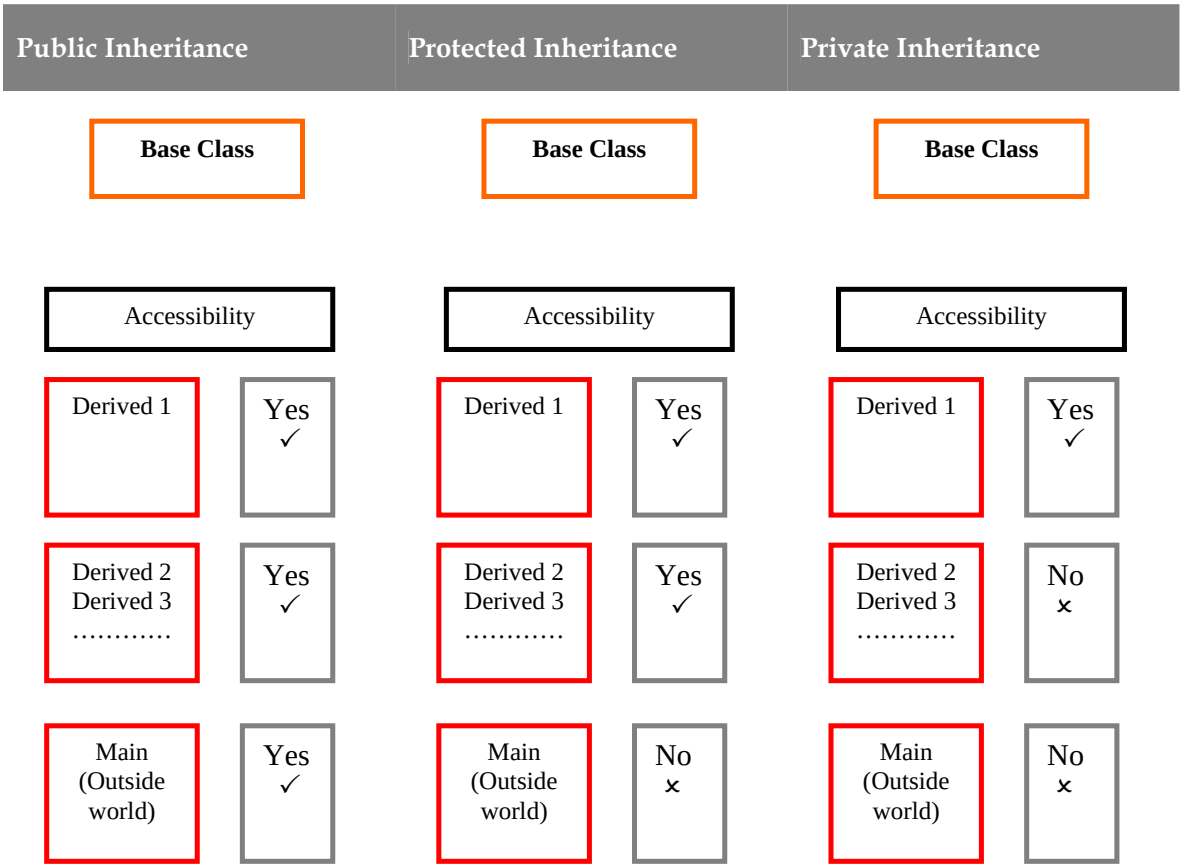
class GrandParent{
};

class Parent: protected GrandParent{
};
class Child: protected Parent {
public:
        Child()
        {
                DoSomething(*this);
        }
};
```

Importance of Private and Protected inheritance:
As we have seen that private and protected inheritance is being used for implementing **"Implemented in terms of"** relationship so it is important that we limit different features of base class in child classes that is what we achieve using either private or protected inheritance according to our requirement.

Comparison of public, protected and private inheritance:

We can show accessibility of public member functions of base class in derived classes in these three different types of inheritance as follows,



Private data members will NOT be accessible in any derived class or in main function.
Protected data members will become private data members in case of private inheritance and protected data members of derived class in case of protected inheritance.

A Good Programming Exercise:
A good programming exercise would be that you write a program that shows accessibility of all types of members functions for all types of inheritance in derived class/es.

© Virtual University of Pakistan

Lecture No.28

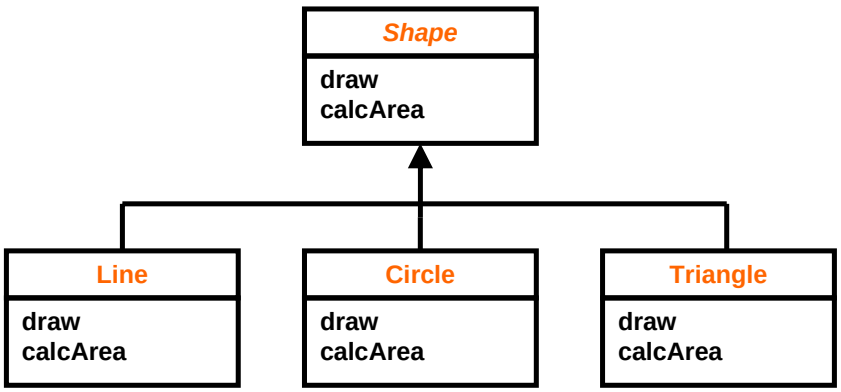
28.1. Virtual Functions

Problem Statement:

*Develop a function that can draw different types of **geometric shapes** from an array*

Shape Hierarchy:

We have to implement following shape hierarchy for this,



Problem Description:

a. We want to implement this shape hierarchy in such a manner that our function will take Shape pointers array (Shape * []) and this array size as parameters and then it will draw the appropriate shape.
b. This Shape pointer array (Shape * []) may store pointers to all kinds of Shapes (Line, Circle and Triangle classes) as these classes have IS-A relationship with Shape class (We know base class pointer can store pointer of any of its derived classes in case of public inheritance IS-A relationship).
c. The purpose of implementing this hierarchy in such a way is that we can avoid complex code of calling draw function for each class separately after checking the class type, instead we can use a single function call in a loop to draw all kinds of Shapes as shown below,

```
void drawShapes(Shapes *array[], int size){

        for (int i = 0 ; i < size ; i ++)
            array[i]->draw(); // this function call will work for all types of
                                    //Shapes (Line, Circle and Triangle)

}
```

Implementation:

It can be done using concept of inheritance as there exists "IS-A" relationship between general Shape class and derived Line, Circle and Triangle classes.

So in code below, we wrote a general Shape class and then we publicly derived Line, Circle and Triangle classes from it, in main we create objects of those Shapes that we needed like Line, Circle and Triangle and stored them in single shape array, then we called **drawShapes** function by passing this array to draw appropriate shapes.

```
class Shape {
        …
protected:
        char _type;
public:
        Shape() { }
        void draw(){ cout << "Shape\n"; }
        int calcArea() { return 0; }
        char getType() { return _type; }
};
class Line : public Shape {

public:
        Line(Point p1, Point p2) {

        }
        void draw(){ cout << "Line\n"; }
};
class Circle : public Shape {
public:
        Circle(Point center, double radius) {

        }
        void draw(){ cout << "Circle\n"; }
        int calcArea() { … }
};
class Triangle : public Shape {

public:
        Triangle(Line l1, Line l2, double angle)
        {
}
        void draw(){ cout << "Triangle\n"; }
        int calcArea() { … }

};
int main() {
        Shape* _shape[ 10 ];
```

 © Virtual University of Pakistan

```
        Point p1(0, 0), p2(10, 10);
        shape[1] = new Line(p1, p2);
        shape[2] = new Circle(p1, 15);
        void drawShapes( shape, 10 );
        return 0;
}

/*Function drawShapes()*/

void drawShapes(Shape* _shape[],   int size) {
        for (int i = 0; i < size; i++) {
                _shape[i]->draw();
        }
}
```

After compiling this code we will see the following output,

Sample Output

```
Shape
Shape
Shape
Shape
…
```

As you have seen this code is not showing correct output, it is showing text Shape again and again instead of showing appropriate class name that was stored in Shape array like Line, Circle or Triangle so there is some logical error in this code.
Where is problem?
Problem is that as we stored our derived classes (Line, Circle and Triangles) Objects in Shape pointer array and then called draw method; every time draw method of Shape class was called.
Why?
Due to the reason that static type of this array is Shape * so draw method is Shape class will always be called whether pointer stored in it is Line, Circle or Triangle pointer or Shape class pointer.

So we have to think about some modification,

Solution 1:

One solution is that we can modify our **drawShapes** function as follows,

```
void drawShapes( Shape* _shape[], int size) {
        for (int i = 0; i < size; i++) {

                // Determine object type with switch & accordingly call draw()
method
```

```
        }
}
```

The required Switch Logic for this kind of functionality is given below,

```
switch ( _shape[i]->getType() )
{
        case 'L':
                static_cast<Line*>(_shape[i])->draw();
                break;
        case 'C':
                static_cast<Circle*>(_shape[i])->draw();
                break;
        …
}
```

Equivalent If Logic is given below,

```
if ( _shape[i]->getType() == 'L' )
        static_cast<Line*>(_shape[i])->draw();
else if ( _shape[i]->getType() == 'C' )
    static_cast<Circle*>(_shape[i])->draw();
…
```

Sample Output:

```
Line
Circle
Triangle
Circle
…
```

You can see that this code is very complex for both switch or if else conditions.

But there are many Problems with this approach these are given below:

Delocalized Code
Suppose we have to write another function that prints area of each shape from an input array. We have to write same switch or if else logic in that function implementation also to work it correctly.

```
void printArea( Shape* _shape[], int size) {
        for (int i = 0; i < size; i++) {
                // Print shape name.
                // Determine object type with
                // switch & accordingly call
                // calcArea() method.
        }
}
```

 © Virtual University of Pakistan

Required Switch Logic

```
switch ( _shape[i]->getType() )
{
        case 'L':
                static_cast<Line*>(_shape[i])->calcArea(); break;
        case 'C':
         static_cast<Circle*>(_shape[i])->calcArea(); break;
        …
}
```

The above switch logic is same as was in function drawArray() with difference of name of function being called.

So this approach will result in delocalized code (same code in different places) with following consequences,
Writing same code again and again at different places may produce errors as programmer may forget to write switch cases.
Suppose we have added one more Shape in our program then we will have to add one more switch case or if else condition in all functions where we have used this logic if programmer by mistake forgets to add it in any single function whole program will show incorrect output.
So due to above mentioned reasons this sort of code is very hard to maintain.

Solution?

To avoid switch, we need a mechanism that can select the message target (class) automatically!

Polymorphism Revisited:

In OO model, polymorphism means that different objects can behave in different ways for the same message (stimulus) consequently, sender of a message does not need to know the exact class of receiver.
In other words when we have inheritance relationship and we have written basic structure of our program correctly we need not to worry about the no. of classes in our program. For example in case of shapes hierarchy (Line, Circle, Triangle and so on…) our program will keep working correctly if we add more shapes, draw method of appropriate shape class should automatically be called, that is the benefit of Object Oriented Programming,

```
        void drawShapes(Shapes *array[], int size){

                for (int i = 0 ; i < size ; i ++)
                    array[i]->draw(); // this function call will work for all types of
                                                //Shapes (Line, Circle and Triangle)

        }
```

But this kind of functionality was not being achieved in above approach in which we had to write one separate case for each type of shape class.

To achieve this kind of functionality we have the concept of **virtual functions** we make those functions virtual in base class which will be implemented by derived classes according to their requirements.

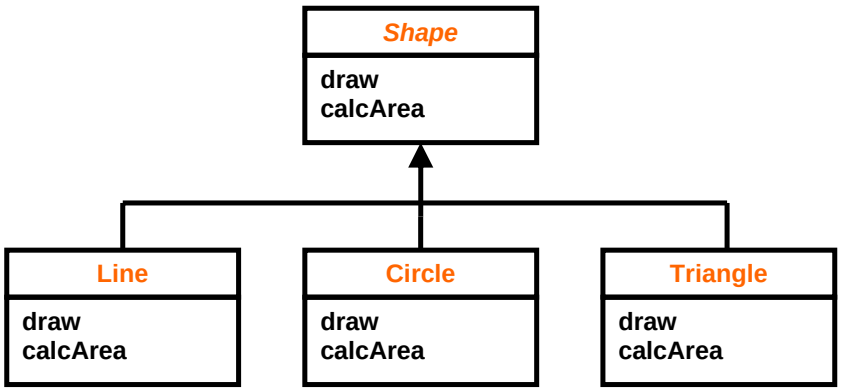
28.2. Virtual Functions:

Virtual functions achieve exactly same kind of functionality that was achieved in above code with complex code of switch statement.

- Target class of a virtual function call is determined at run-time **automatically**.
- In C++, we declare a function virtual by preceding the function header with keyword "virtual"

Now we see how we can use virtual functions in case of our shape hierarchy,

28.3. Shape Hierarchy



We will make those functions virtual in shape that need to be overridden by derived classes like Draw method,

```
class Shape {
                …
                virtual void draw();
        };
```

```
class Shape {
        …
        virtual void draw();
        virtual int calcArea();
};
class Line : public Shape {
        …
        virtual void draw() { cout<<"Line…\n";
                }
};
```


© Virtual University of Pakistan

```
class Circle : public Shape {
        …
        virtual void draw() { cout<<"Circle….\n";
                    }
        virtual int calcArea();
};
class Triangle : public Shape {
        …
        virtual void draw(){ cout<<"Triangle…\n";
                    }

        virtual int calcArea();
};
void drawShapes(Shape* _shape[], int size) {
                for (int i = 0; i < size; i++) {
                        _shape[i]->draw();
        }
}
```

Sample Output:

```
Line
Circle
Triangle
Circle
…
```

Similarly if we have another function that will calculate and print area of different shapes it will also be written in same way, (Now we have no need to add switch or if else code)

```
void printArea(Shape* _shape[], int size) {
                for (int i = 0; i < size; i++) {
                        // Print shape name
                        cout<< _shape[i]->calcArea();
                        cout << endl;
        }
}
```

28.4.Static vs Dynamic Binding

- **Static binding** means that target function for a call is selected at compile time
- **Dynamic binding** means that target function for a call is selected at run time

```
Line _line;
_line.draw();  // Always Line::draw will be called
```

Shape* _shape = new Line();
_shape->draw(); // Shape::draw called if draw() is not virtual because of static type of Shape *

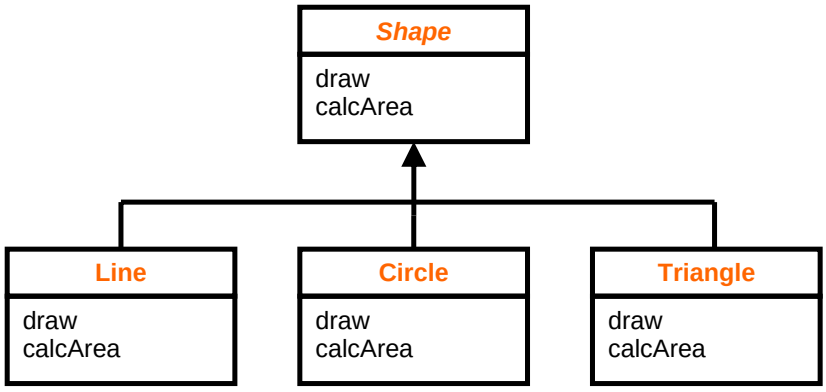
Shape* _shape = new Line();
_shape->draw(); // Line::draw called as draw() is virtual in base Shape class

© Virtual University of Pakistan

Lecture No.29

Previous Lecture Review:

The main concept of Polymorphism is that the same method can behave differently according to the object with respect of which it has been called.
We combine all related classes having same kind of functionality in a class hierarchy as shown below,



The draw and calcArea methods here are showing polymorphic behavior as they exist in base as well as in derived classes, the method that will be called is decided at runtime according to the nature of the calling object, for example in the case of above class hierarchy if we call draw method with respect of Line class object, draw of Line class will be called and if we call it with reference of Circle or Triangle class, draw method of Circle or Triangle class will be called.

We achieve this type of polymorphism[13] using virtual functions.
The advantage of this approach is that sender simply pass method call and appropriate method is called automatically, as we saw in previous lecture in which we simply used for loop to draw all kinds of shapes using shape pointers array.
We also saw the drawbacks of achieving same functionality without virtual functions using switch statement that resulted in delocalized and complex code. Then we saw solution of this problem using virtual functions.

29.1. Abstract Classes

In our Shape class hierarchy Shape class has no real world existence, there will be no object in real world with name **Shape** there will always be an object of kind of shape like **Line**, **Circle** or **Triangle**, this kind of class with no specific real world existence are called **Abstract Classes**, as these classes are Abstract their behavior becomes also abstract like Draw and calcArea methods in case of Shape class, the actual concepts

[13] Polymorphism may exist in same class (function overloading) or it may exist in different classes (function overriding) here we are referring to function overriding (same functions in base and derived classes).

are realized in derived classes, like all kinds of shapes will have draw and calcArea method, so we add these methods in general Shape class and all kinds of Shapes inheriting for this general shape class given their own implementation of these methods.

Abstract class's objects cannot be instantiated they are used for inheriting interface and/or implementation, so that derived classes can give implementation of these concepts.

29.2. Concrete Classes

Concrete classes Implements a concrete concept they can be instantiated they may inherit from an abstract class or another concrete class. So far the classes we studied were concrete classes.

29.3. Abstract Classes in C++

In C++, we can make a class abstract by making its function(s) pure virtual. Conversely, a class with no pure virtual function is a concrete class (which object can be instantiated)

29.4. Pure Virtual Functions

A pure virtual represents an abstract behavior and may have not implementation for example draw method in Shape class represent abstract behavior as Shape class itself doesn't have its existence in real world so there is no question of drawing it however its derived concrete classes like Line, Circle and Triangle does have physical existence and overridden draw method in these classes will have implementation . A function is declared pure virtual by following its header with "= 0".

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

A class having pure virtual function(s) becomes abstract

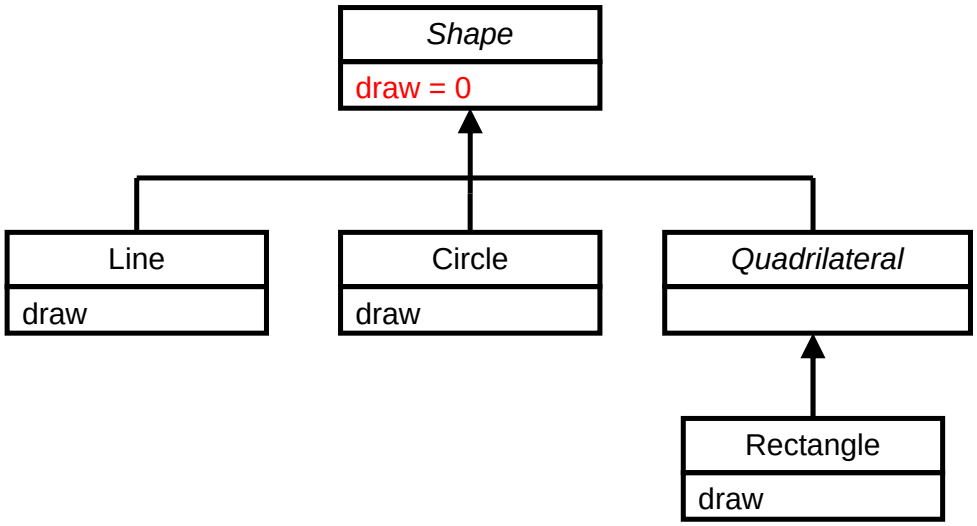
```
class Shape {
        …
public:
        virtual void draw() = 0;
};
```

Now when we will try to created object of our Shape class in our program compiler will give an error as shown below,

```
Shape s;                // Error!
```


© Virtual University of Pakistan

29.5.Shape Hierarchy



A derived class of an abstract class remains abstract until it provides implementation for all pure virtual functions as shown below in other words we can say that at least one class in bottom of class hierarchy should give implementation of pure virtual function (Abstract classes are present at root or near root of the class hierarchy tree, whereas concrete classes are near leaves of class hierarchy tree) see the code below of above class heirarchi here Quadrilateral is also abstract class as it is derived from Shape class but is not giving implementation of draw method we can also not create its object and it is necessary to have one more derived class from quadrilateral that gives implementation of draw method otherwise there will be compiler error as we can not have all abstract classes in heirarchi there should be at least one concrete class at leaf in class heirarchi,

```
class Quadrilateral : public Shape {
        …
        // No overriding draw() method
};
…
Quadrilateral q; // Error!

class Rectangle:public Quadrilateral{
        …
        public:
        // void draw()
        virtual void draw() {      // once a function is declared as virtual it remains
virtual in all hierarchy
                … // function body
        }
};
…
Rectangle r;              // OK
```

29.6. Virtual Destructors

The destructor is called according to static type of any class pointer for example if we have saved derived class pointers in shape class pointers array as we did in previous lecture when we will call destructor using delete operator the destructor of base class Shape will be called as static type of array is Shape. This will destroy the base class object only derived class object will not be destroyed this is explained in the example code below,

```
class Shape {
        …
        public:
        ~Shape() {
                cout << "Shape destructor
        called\n";
        }
};

class Quadrilateral : public Shape {
        …
        public:
        ~Quadrilateral() {
                cout << "Quadrilateral destructor
        called\n";
        }
};

class Rectangle : public Quadrilateral {
        …
        public:
        ~Rectangle() {
                cout << "Rectangle destructor
                called\n";
        }
};

int main() {
        Shape* pShape = new Rectangle();
        delete pShape;
        return 0;
}
```

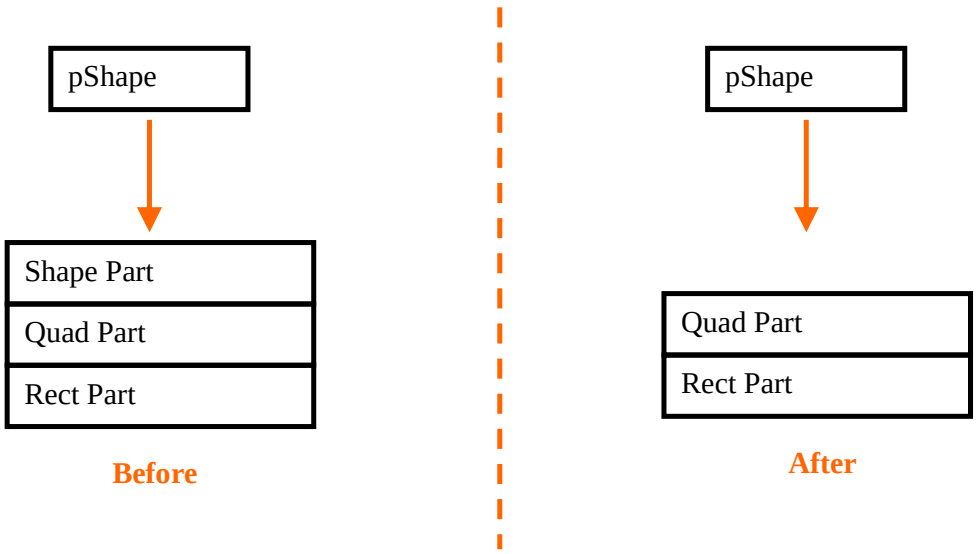
When delete operator is applied to a base class pointer, base class destructor is called regardless of the object type
Output

```
Shape destructor called
```


© Virtual University of Pakistan

Result

You can see by diagram below that only base class part of object will be deleted other parts will remain as it is this result in memory leak (wastage of memory),



This issue will be solved in the same way as we solved the problem of calling derived classes **Draw methods** using base class pointer by the use **Virtual keyword**.

Virtual Destructors

Make the base class destructor virtual as we made Draw method virtual in base class,

```
class Shape {
        …
        public:
        virtual ~Shape() {
                cout << "Shape destructor  called\n"; }
};
class Quadrilateral : public Shape {
        …
        public:
        virtual ~Quadrilateral() {
                cout << "Quadrilateral destructor called\n";
        }
};
class Rectangle : public Quadrilateral {
        …
        public:
        virtual ~Rectangle() {
                cout << "Rectangle destructor called\n";
```

```
        }
};
```

Now base class destructor will run after the derived class destructor

```
int main() {
        Shape* pShape = new Recrangle();
        delete pShape;
        return 0;
}
```

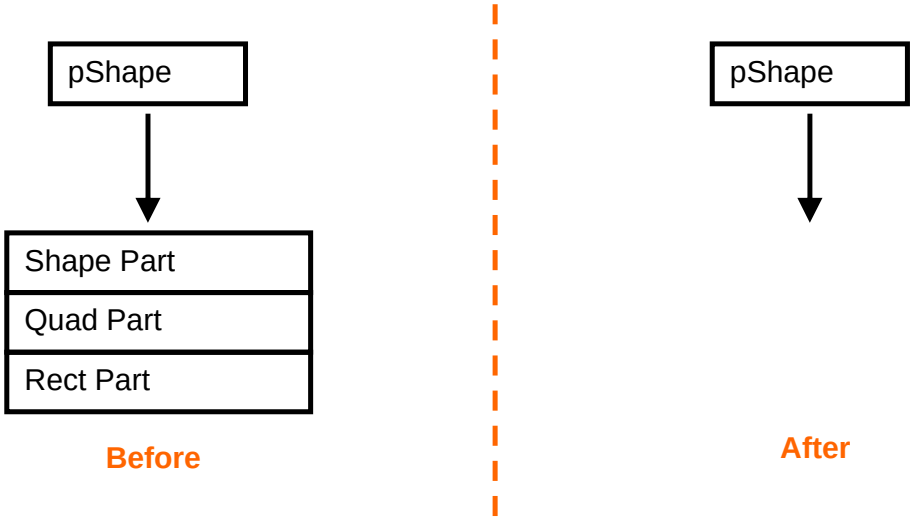
Output

Now you can see that output is correct all destructors are being called in correct order,

```
Rectangle destructor called
Quadrilateral destructor called
Shape destructor called
```

Result

Now you can see result complete object is being deleted so there is no memory leak (waste of memory),



29.7. Virtual Functions – Usage

Virtual function are used in two ways,

© Virtual University of Pakistan

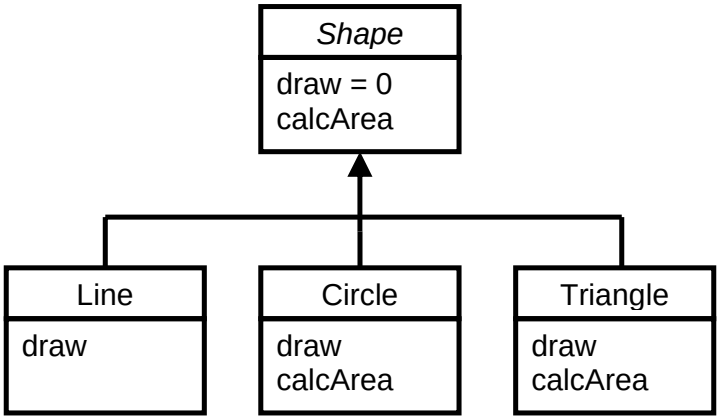
- Virtual Functions can be used to when we want to inherit interface and implementation (Simple virtual functions) mean base class as well as derived class will have implementation.

  virtual void draw();
- Just inherit interface (Pure Virtual functions) mean only derived classes will will have implementation base may not have implementation.

  virtual void draw() = 0;

Inherit interface and implementation:

First case of simple virtual functions is useful when we have some derived classes that will also not have implementation of virtual method for example Line is also also Shape but it doesn't have area similarly if we had Point derived class it also doesn't have any area in both cases we will simply not write implementation of calcArea() method and hence calcArea of base Shape class will be called which will simply display area as zero this is shown below,




```
class Shape {
…
        virtual void draw() = 0; // pure virtual functions

        virtual float calcArea() {  // simple virtual functions
                return 0;
        }
};
```

So as **calcArea()** is simple virtual function so, each derived class of **Shape** inherits default implementation of **calcArea(), s**ome classes may override this, such as **Circle** and **Triangle** and Others may not, such as **Point** and **Line.**

Inherit interface only:

We have made draw method as pure virtual function because each shape will need to draw whether it is simple point or line or any other shape so its suitable to be decalred as pure virtual,

As **draw()** is Pure Virtual Function so, each derived class of **Shape** inherits interface (prototype) of **draw(), e**ach concrete derived class has to provide body of **draw()** by overriding it.

29.8.V Table

Now we see compiler keeps track of virtual functions and call them correctly according to nature of the object with respect to which they are being called,

Compiler builds a virtual function table (vTable) for ***each class having virtual functions***

A vTable contains a pointer for each virtual function,

Pointer to 1st virtual function
Pointer to 2nd virtual function
….....
…….
…….

V Table

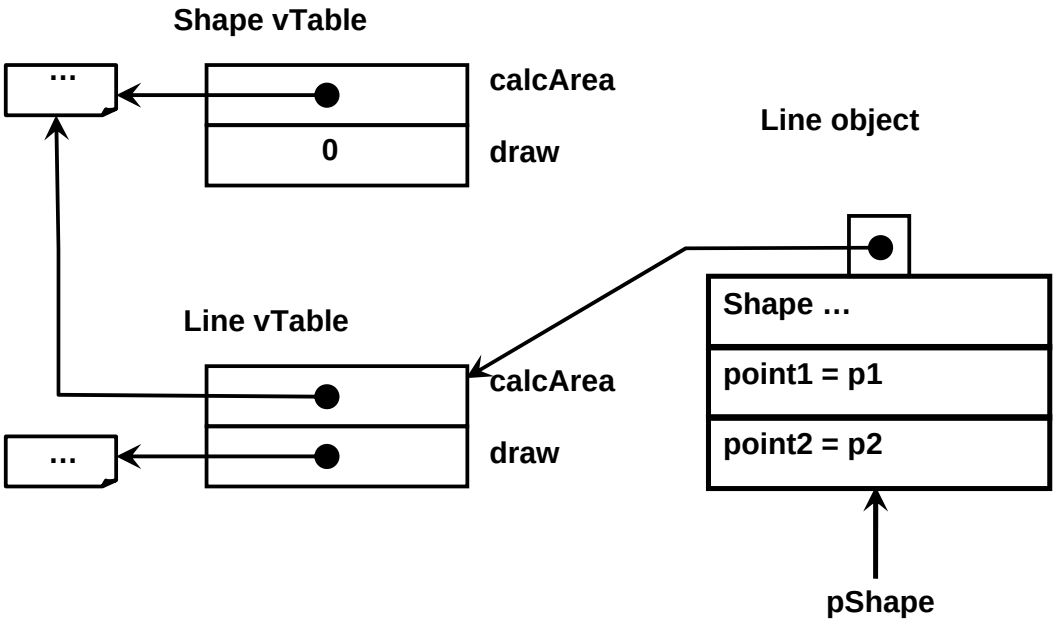
Consider the code below to see how virtual tables are created,

```
int main() {
        Point p1( 10, 10 ), p2( 30, 30 );
        Shape* pShape;

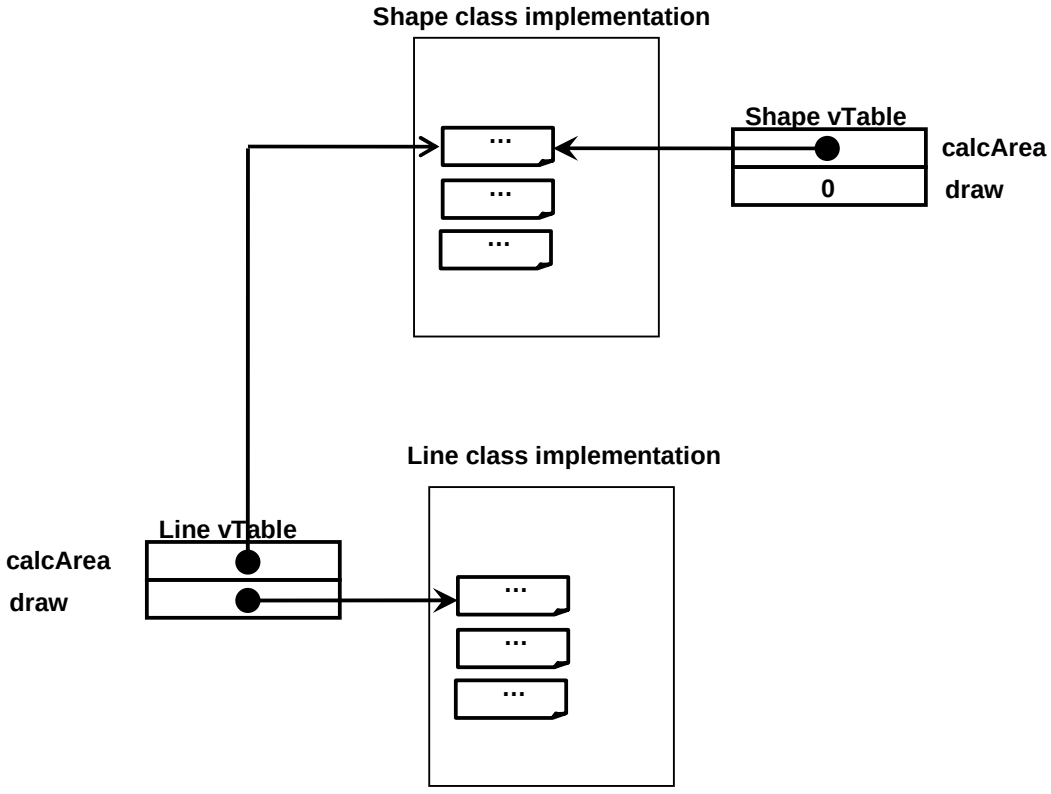
        pShape = new Line( p1, p2 );
        pShape->draw();
        pShape->calcArea();
        return 0;
}
```

We are creating Line class object here and storing its refernence in Shape class pointer and then polymorphically calling methods draw and calcArea as Shape pointer is having Line class object reference so Line class method draw and calcArea will be called we see how this actually happens,

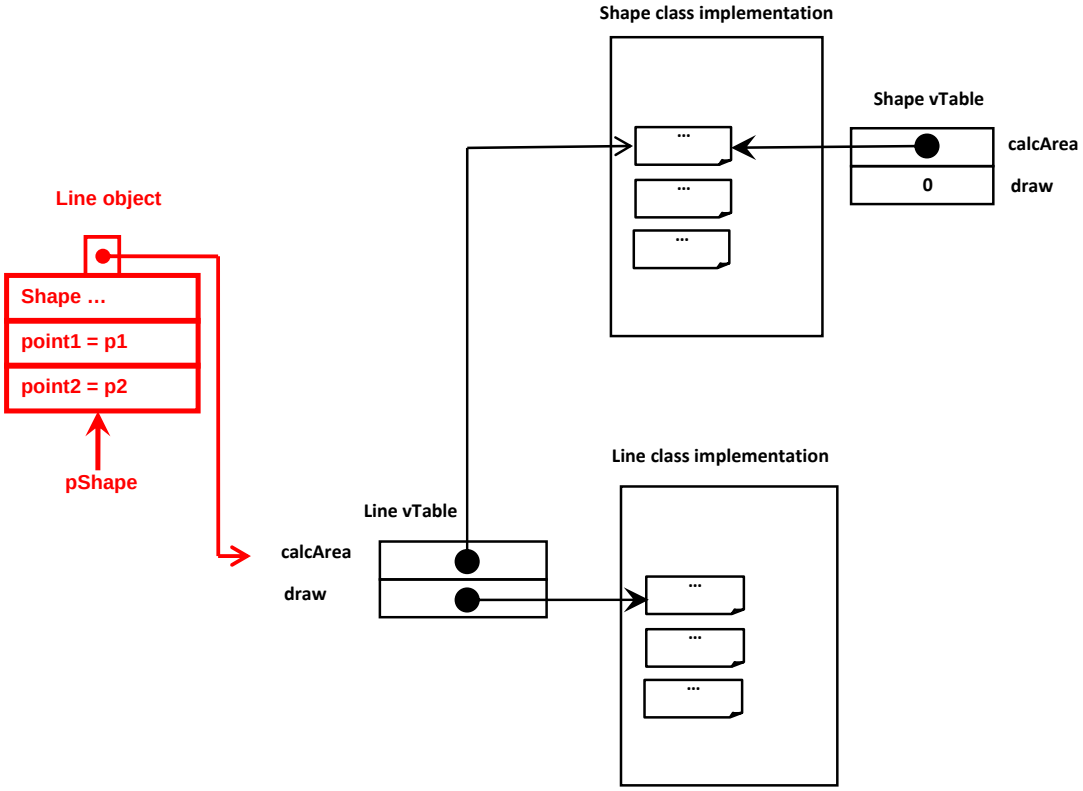
 © Virtual University of Pakistan



First of all when this code will be compiled v tables along with implementation code of virtual and non virtual functions for all classes will be generated by compiler as shown below,



When line object will be crated in main it will also have pointer to v table of Line class as shown below,



29.9. Dynamic Dispatch (Dynamic Binding)

It happens in case of virtual functions, for non-virtual functions, compiler just generates code to call the function.

In case of virtual functions, compiler generates code to

- access the object
- access the associated vTable
- call the appropriate function

Conclusion

Virtual Functions should be added in code with care because they add,

- Memory overhead due to V-Tables
- Processing overhead due to extra pointer manipulation

However, this overhead is acceptable for many of the applications.

Moral: "Think about performance requirements before making a function virtual".

© Virtual University of Pakistan

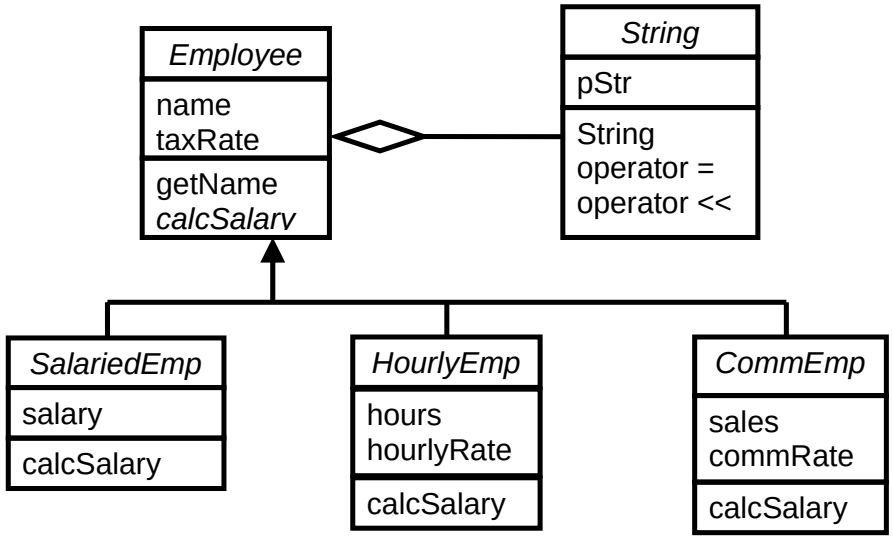
Lecture No.30

30.1.Polymorphism – Case Study: A Simple Payroll Application

We have studied polymorphism implementation in detail using virtual functions now we see an example showing how it may be useful for us,

Problem Statement
Develop a simple payroll application for a company; there are three kinds of employees in the system: salaried employee, hourly employee, and commissioned employee. The system should take input as an array containing employee objects, calculates salary polymorphically (according to employee object), and generates report.

OO Model



You can see that this model is very similar to Shape class hierarchy we saw previously, its implementation in c++ is shown below,

Class *Employee*

```
class Employee {
        private:
                String name;
                double taxRate;
        public:
                Employee( String&, double );
                String getName();
                virtual double calcSalary() = 0;
};
Employee::Employee( String& n, double tr ): name(n){
 taxRate = tr;
}
```

```
String Employee::getName() {
        return name;
}
```

Class SalariedEmp

```
class SalariedEmp : public Employee
{
private:
                double salary;
public:
                SalariedEmp(String&,double,double);
                 virtual double calcSalary();
};
SalariedEmp::SalariedEmp(String& n, double tr, double sal) : Employee( n, tr ) {
        salary = sal;
}
double SalariedEmp::calcSalary() {
        double tax = salary * taxRate;
        return salary – tax;
}
```

Class HourlyEmp

```
class HourlyEmp : public Employee {
private:
        int hours;
        double hourlyRate;
public:
        HourlyEmp(string&,double,int,double);
        virtual double calcSalary();
};
HourlyEmp ::HourlyEmp( String& n, double tr, int h, double hr ) : Employee( n, tr
) {
        hours = h;
        hourlyRate = hr;
}
double HourlyEmp::calcSalary()
{
        double grossPay, tax;
        grossPay = hours * hourlyRate;
        tax = grossPay * taxRate;
        return grossPay – tax;
}
```

 © Virtual University of Pakistan

Class CommEmp

```
class CommEmp : public Employee
{
private:
        double sales;
        double commRate;
public:
        CommEmp( String&, double, double, double );
        virtual double calcSalary();
};
CommEmp::CommEmp( String& n, double tr, double s, double cr ) : Employee(
n, tr ) {
        sales = s;
        commRate = cr;
}
double CommEmp::calcSalary()
{
        double grossPay = sales * commRate;
        double tax = grossPay * taxRate;
        return grossPay – tax;
}
```

A Sample Payroll

```
int main() {
        Employee* emp[10];
        emp[0] = new SalariedEmp( "Aamir", 0.05, 15000 );
        emp[1] = new HourlyEmp( "Faakhir", 0.06, 160, 50 );
        emp[2] = new CommEmp( "Fuaad", 0.04, 150000, 10 );
        …
        generatePayroll( emp, 10 );
        return 0;
}
```

Function that takes Empolyee poinsters array and calls appropriate getName and calcSalary methods

```
void generatePayroll(Employee* emp[], int size) {

        cout << "Name\tNet Salary\n\n";
        for (int i = 0; i < size; i++) {
        cout << emp[i]->getName() << '\t' << emp[i]->calcSalary() << '\n';

        }
}
```

Sample Output

Name	Net Salary
Aamir	14250
Fakhir	7520
Fuaad	14400
…	

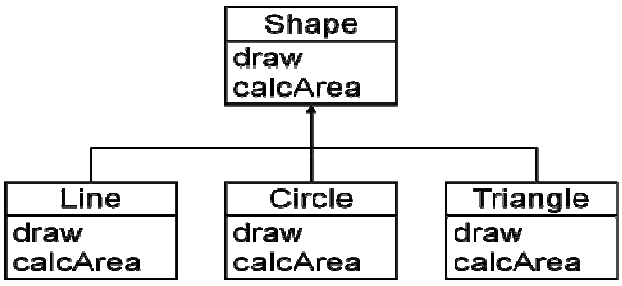
Output is as required displaying employee names and their salary polymorphically.

Important point to note here is that *Polymorphism always works with pointers of class objects not with actual objects*. In above example we used base Employee pointer array to store pointers of derived classes of class Employee and then we polymorphically called generatePayroll method to call getName and calcSal methods in a loop.

Never Treat Arrays Polymorphically:

If we use arrays of Objects Polymorphically then problem occurs as shown below,

30.2. Shape Hierarchy Revisited:



We have not made Shape class Abstract so that we can create its objects to show the point that calling array of objects polymorphically results in errors,
See the shape hierarchy again,

```
class Shape {
        …
public:
        Shape();
        virtual void draw(){
                cout << "Shape\n";
        }
        virtual int calcArea() { return 0; }
};

class Line : public Shape {
        …
```

 © Virtual University of Pakistan

```
public:
        Line(Point p1, Point p2);
        void draw(){ cout << "Line\n"; }
};

void drawShapes( Shape _shape[], int size ) {
        for (int i = 0; i < size; i++) {
                _shape[i].draw();
        }
}

//Polymorphism & Arrays

int main() {
        Shape _shape[ 10 ];
        _shape[ 0 ] = Shape();
        _shape[ 1 ] = Shape();
        …
        drawShapes( _shape, 10 );
        return 0;
}
```

In above code we created array of ten objects of Shape class and then passed it to drawShapes function, note that in above code we passed Shapes array (Shapes []) instead of Shapes pointers array (Shapes * []) to drawShapes function.
The code works draw method for each Object of Shape class is called and output is displayed below,

Sample Output

```
Shape
Shape
Shape
…
```

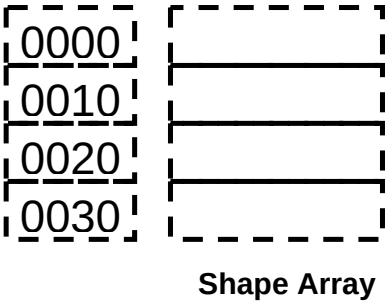
Now we try to do polymorphism here by creating lines objects and passing to this functions as we did previously in case of pointers with the difference that this time we are passing objects array instead of pointers array.

```
int main() {
        Point p1(10, 10), p2(20, 20), …
        Line _line[ 10 ];
        _line[ 0 ] = Line( p1, p2 );
        _line[ 1 ] = Line( p3, p4 );
        …
        drawShapes( _line, 10 );
        return 0;
}
```

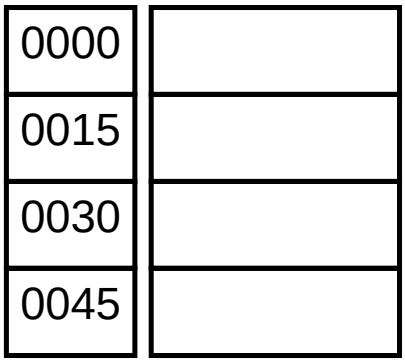
Sample Output:

```
Shape
// Run-time error
```

This time also program is compiled correctly but when we executed it only Shape text is displayed for first object of array and then program terminates abnormally.
The reason behind this output is,
When we passed array of Line objects as parameter this array was converted implicitly to array of Shapes objects by compiler as Line IS A kind of shape, same thing was happened with pointers of array objects but the difference now is that this Line objects array will be treated as Shape objects array, so draw method of first shape object is called and it displays text Shape, for next iteration of loop when compiler calculated address of next object it did it with respect to Shape objects but the actual array consists of Line objects so compiler will incorrectly calculate address of next object as size of Shape class object and Line class objects are different and when it will call draw method with incorrect address of object program will terminate abnormally this is give in diagram below,



Shape Array



Line Array

```
_shape[ i ].draw();
*(_shape + (i * sizeof(Shape))).draw();
```

As shown above compiler will do calculations with respect to Shape array and will calculate next object address as 0010 because it is treating Line array as Shape array but actually next Line object is present at address 0015 so runtime error will be generated as there is no new object at address 0010 and program will terminate abnormally.

Original drawShapes() method that was using pointers is given below

```
void drawShapes(Shape* _shape[],  int size) {
        for (int i = 0; i < size; i++) {
```

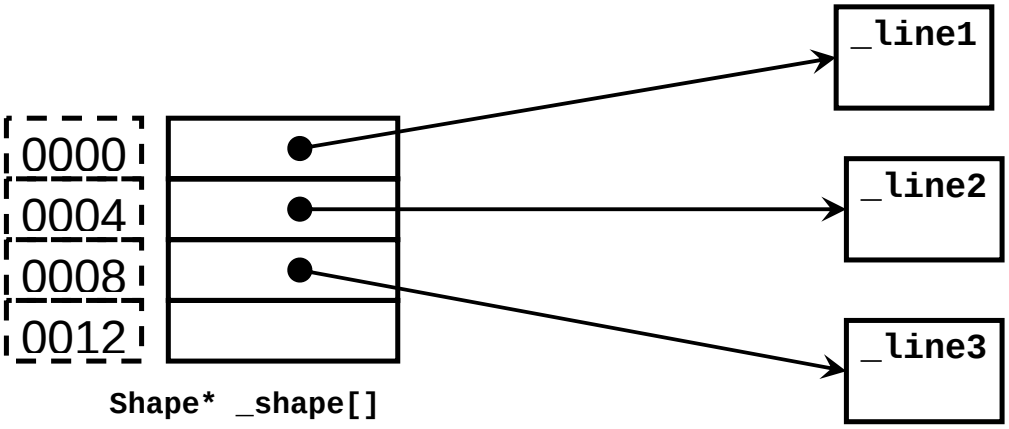
 © Virtual University of Pakistan

```
            _shape[i]->draw();
        }
}
```

It shows correct output,
Sample Output:

```
Line
Line
Line
…
```

Because in case of pointers array, the size of each entry is same as each entry contains a pointer and all pointers are of 4 bytes in c++ whatever they are point to Shape class or to Line class or any other built in or user defined type. So in this case whether our array contain Shape * or Line * next object address will be present after 4 bytes, so our drawShapes function will execute correctly,




```
_shape[i]->draw();
(_shape + (i * sizeof(Shape*)))->draw();
```

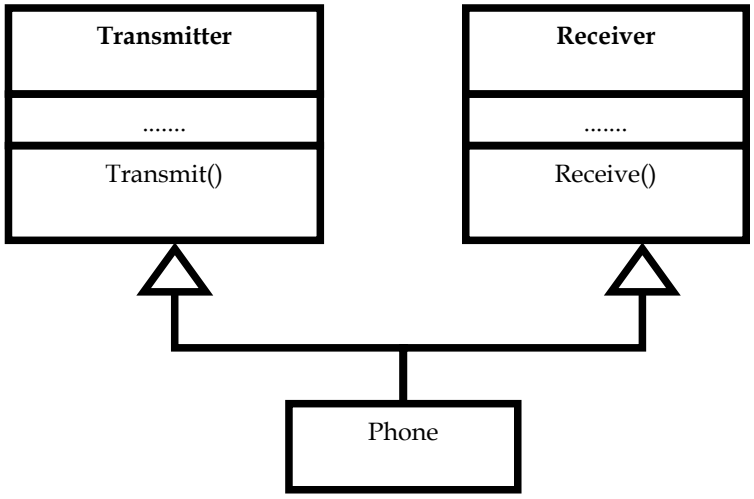
So moral of the story is Never use arrays polymorphically because location of elemetns in any array is calculated using array type and in polymorphism we have the liberty of changing child array to parent array that will result in erraneous calcualation of location of elements in array.

Lecture No.31

31.1.Multiple Inheritance

We have seen multiple inheritance in start of OOP now we see its implementation in C++, A class in C++ can inherit from more than one classes like Phone class can inherit from Transmitter or Receiver.

Examples:




```
class Phone:    public Transmitter, public Receiver
// As phone class is publicly inherited so any class derived from Phone class will also have access to public and protected members of Transmitter and Receiver class
{
...
};
```

Derived class can inherit from public base class as well as private and protected base classes

```
class Mermaid: private Woman, private Fish
// As Mermaid class is privately inherited from Woman and Fish so any class derived from Mermaid class will Not have access to public and protected members of Woman  and Fish classes

{
...
};
```

Multiple Inheritance

The derived class inherits data members and functions from all the base classes
Object of derived class can perform all the tasks that an object of base class can perform

 © Virtual University of Pakistan

Example

```
int main(){
        Phone obj;
        obj.Transmit(); // method of Transmitter class
        obj.Receive(); // method of Receiver class
        return 0;
}
```

Multiple Inheritance

When using public multiple inheritance, the object of derived classes can replace the objects of all the base classes
Example

```
int main(){
        Phone obj;
        Transmitter * tPtr = &obj;
        Receiver * rPtr = &obj;
        return 0;
}
```

However note the following,

The pointer of one base class cannot be used to call the function of another base class
The functions are called based on static type

Example:

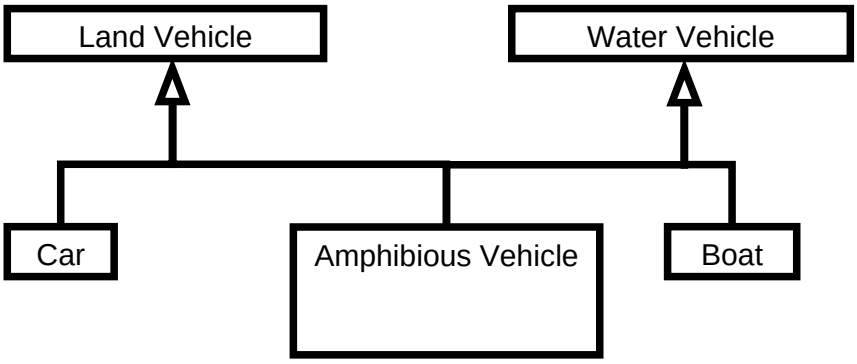
```
int main(){
        Phone obj;
        Transmitter * tPtr = &obj;
        tPtr->Transmit();
        tPtr->Receive();        //Error
        return 0;
}
```

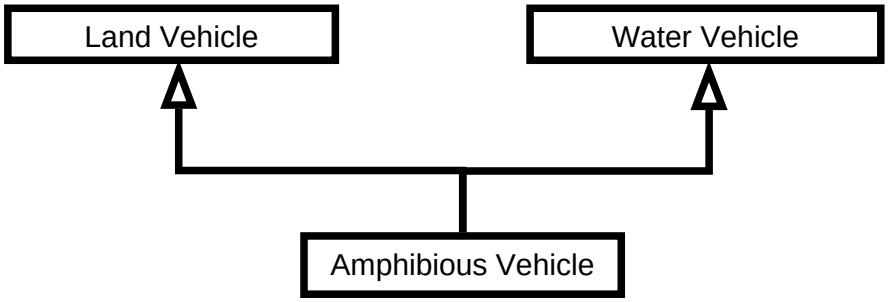
```
int main(){
        Phone obj;
        Receiver * rPtr = &obj;
        rPtr->Receive();
        rPtr->Transmit();       //Error
        return 0;
}
```

31.2.Problems in Multiple Inheritance

- If more than one base class have a function with same signature then the child will have two copies of that function.
- Calling such function will result in ambiguity.

Example:






```
class LandVehicle{
public:
        int GetMaxLoad();
};
class WaterVehicle{
public:
        int GetMaxLoad();
};
class AmphibiousVehicle: public LandVehicle, public WaterVehicle {
};
int main(){
        AmphibiousVehicle obj;
        obj.GetMaxLoad();           // Error
        return 0;
}
```

 © Virtual University of Pakistan

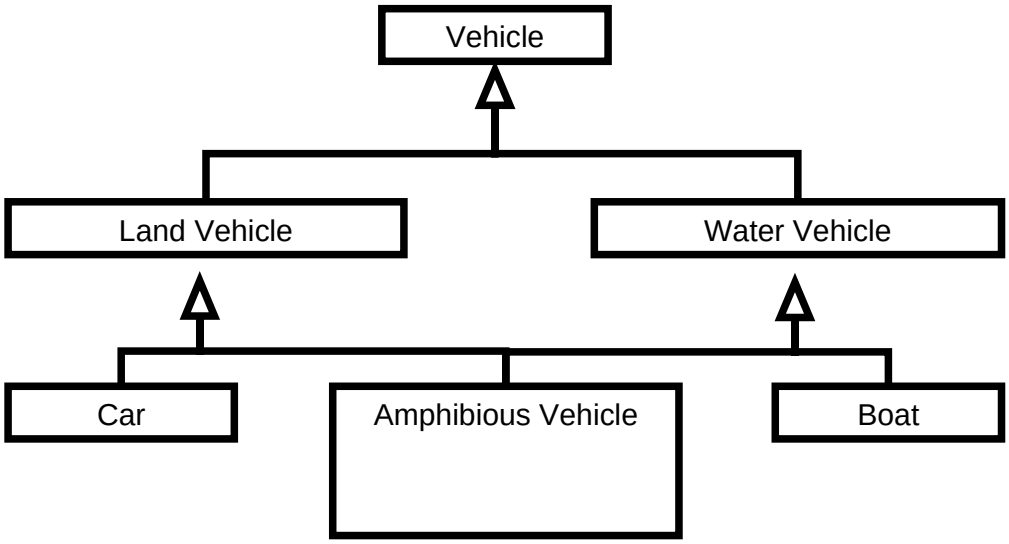
The solution of this problem is that Programmer must explicitly specify the class name when calling ambiguous function

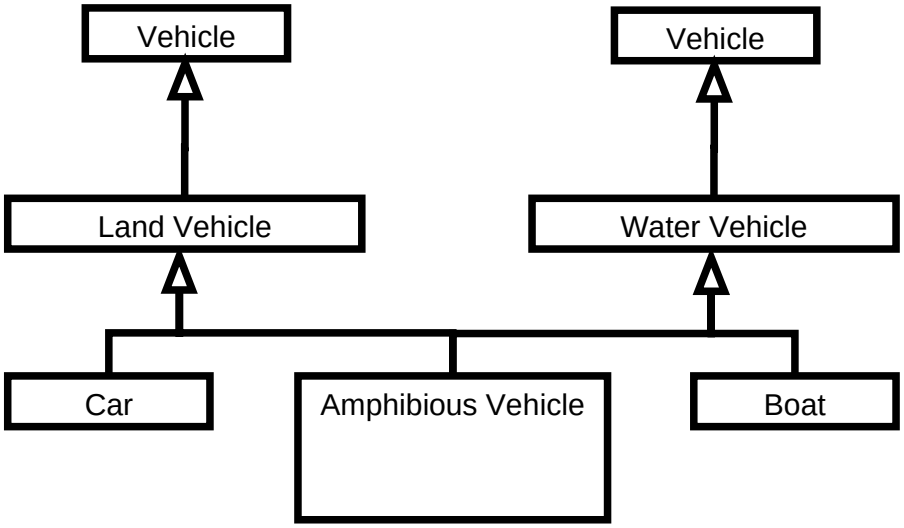
```
int main(){
        AmphibiousVehicle obj;
        obj.LandVehicle::GetMaxLoad();
        obj.WaterVehicle::GetMaxLoad();
        return 0;
}
```

Multiple Inheritance

The ambiguous call problem can arise when dealing with multiple level of multiple inheritance

Example:



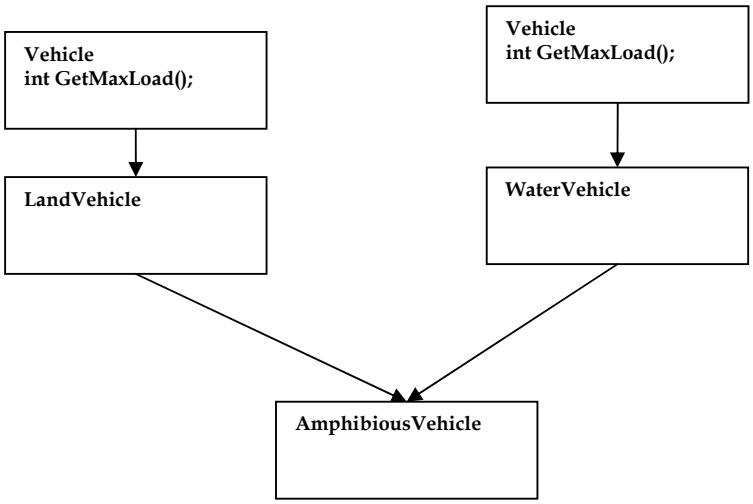



```
class Vehicle{
public:
        int GetMaxLoad();
};
class LandVehicle : public Vehicle{
};
class WaterVehicle : public Vehicle{
};
class AmphibiousVehicle: public LandVehicle, public WaterVehicle {
};

int main(){
        AmphibiousVehicle obj;
        obj.GetMaxLoad();           // Error
        return 0;
}
```

The error is due to the reason that object of AmphibiousVehicle class has two implicit Vehicle class objects one with respect to LandVehicle and one with respect to WaterVehicle class,

 © Virtual University of Pakistan



When call to GetMaxLoad method is made, compiler gets confused which copy of this method should be called.
When we try to remove this error by mentioning that this method belongs to Vehicle class as shown below; error remained as it is due to above mentioned fact.

Example

```
int main()
{
        AmphibiousVehicle obj;
        obj.Vehicle::GetMaxLoad();  //Error
        return 0;
}
```

The reason is same that Vehicle is accessible through two paths, we can avoid this error by explicity mentioning the name of intermediate base class LandVehicle or WaterVehicle with respect to which we want to call this GogMaxLoad method as shown below,

Example

```
int main(){
        AmphibiousVehicle obj;
        obj.LandVehicle::GetMaxLoad();
        obj.WaterVehicle::GetMaxLoad();
        return 0;
}
```

Due to the same reason as mentioned above Data member must be used with care when dealing with more than one level on inheritance.

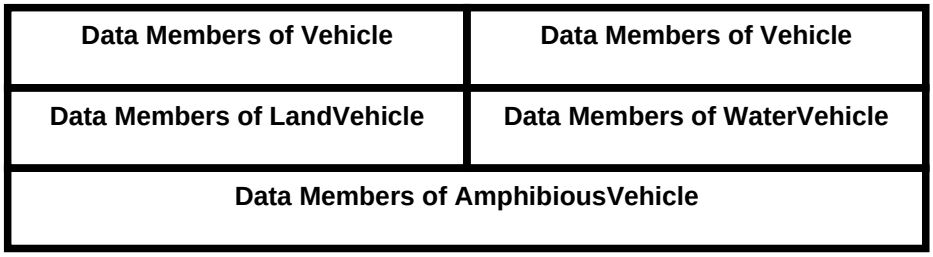
Example

```
class Vehicle{
protected:
```

```
        int weight;
};
class LandVehicle : public Vehicle{
};
class WaterVehicle : public Vehicle{
};
class AmphibiousVehicle:
                        public LandVehicle,
                        public WaterVehicle{
public:
         AmphibiousVehicle(){
                LandVehicle::weight = 10;
                WaterVehicle::weight = 10;
         }
};
```

Here AmphibiousVehicle object has multiple copies of data member **weight**.

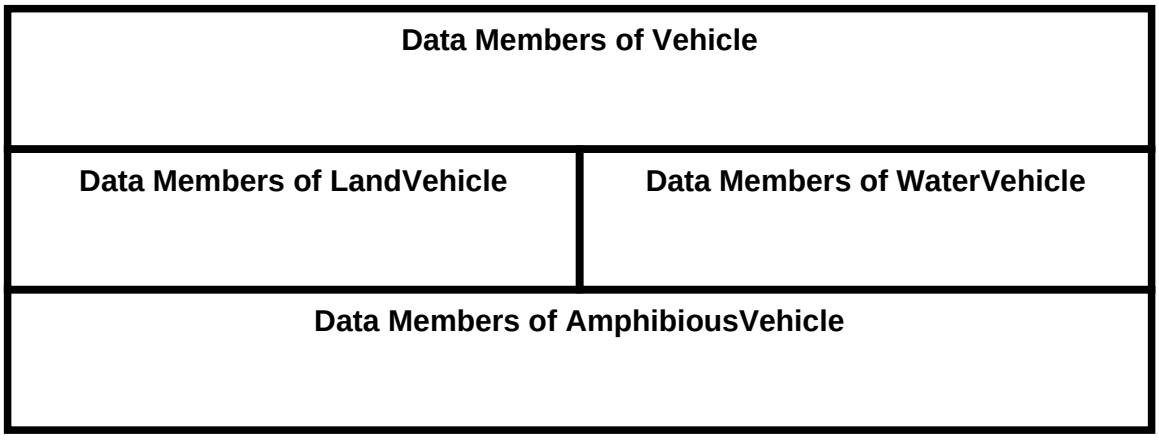
Memory View:



31.3. Virtual Inheritance

The solution to avoid this problem is virtual inheritance so that in multiple inheritance only one copy of base class is generated as shown below instead of two separate copies.

Memory View:



 © Virtual University of Pakistan

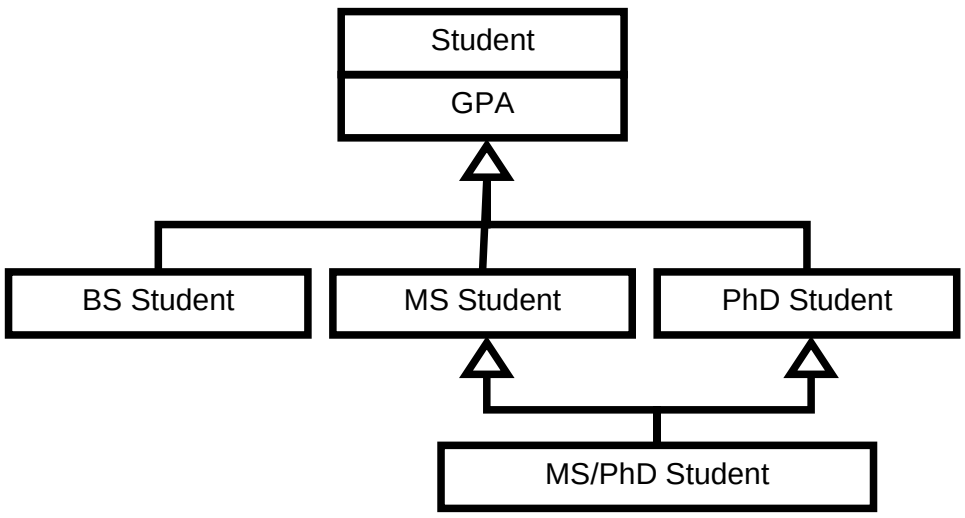
In virtual inheritance there is exactly one copy of the anonymous base class object.
Example:

```
class Vehicle{
protected:
        int weight;
};
class LandVehicle : public virtual Vehicle{
};
class WaterVehicle : public virtual Vehicle{
};
Example
class AmphibiousVehicle: public LandVehicle, public WaterVehicle {
public:
         AmphibiousVehicle(){
                weight = 10;
        }
};
```

When to use Virtual Inheritance?

Virtual inheritance must be used when necessary. It can be used in the situations when programmer wants to use two distinct data members inherited from base class rather than one.

Example



Lecture No.32
Generic Programming

Motivation:

Following function prints an array of integer elements:

```
void printArray(int* array, int size)
{
        for ( int i = 0; i < size; i++ )
                cout << array[ i ] << ", ";
}
```

If we want to print an array of character elements we will write similar function again with different parameters as shown below:

```
void printArray(char* array, int size)
{
        for ( int i = 0; i < size; i++ )
                cout << array[ i ] << ", ";
}
```

similarly for double array we can write,

```
void printArray(double* array, int size)
{
        for ( int i = 0; i < size; i++ )
                cout << array[ i ] << ", ";
}
```

Same will be the case for array of float, short, long and so on….

You can see all these functions are doing same kind of functionality on different data types and it looks repeated task for all data types (basically it is ***function overloading***)

Now if we want to change the way function prints the array. e.g.
from **1, 2, 3, 4, 5**
to **1 2 3 4 5** or **'1' ' 2' '3' '4' '5'** or **"1" "2" "3" "4" "5"**

We will have to change the code in all functions for all data types.

Now consider the **Array** class that is developed to overcome the shortcomings of C++ built in arrays. In C++ arrays, there is no check for array bounds we can insert element at 100th position in an array of size 10 only, our program will compile correctly however this will result in abnormal termination of the program on execution.

```
class Array {
                int* pArray;
                int size;
        public:
```

 © Virtual University of Pakistan

```
        ...
};
```

The above class has been written for integer arrays. Now, if we want to write the same kind of class for all other data types like double or boolean (true/false), the code will be written as shown below,

```
class Array {
        double* pArray;
        int size;
    public:
        ...
};

class Array {
        bool* pArray;
        int size;
    public:
        ...
};
```

This also looks like repeated code and adds effort to write code. Secondly if we want to add a function sum in **Array** class, we have to change all the three classes.

Very imp Mcqs 304&304p

We want some mechanism that enables us to write **single function or class** that works for all data types this technique of writing programs is called ***Generic Programming***.

32.1.Generic Programming

Generic programming refers to programs containing generic abstractions (general code that is same in logic for all data types like printArray function), then we instantiate that generic program abstraction (function, class) for a particular data type, such abstractions can work with many different types of data.

Advantages

Major benefits of this approach are:

Imp for mcq

a. Reusability: Code can work for all data types.
b. Writability: Code takes lesser time to write.
c. Maintainability: Code is easy to maintain as changes are needed to be made in a single function or class instead of many functions or classes.

32.2.Templates

In C++ generic programming is done using templates.

Templates are of two kinds,

a. **Function Templates** (in case we want to write general function like printArray)
b. **Class Templates** (in case we want to write general class like Array class)

Compiler generates different type-specific copies from a single template.
This concept is similar to concept of making prototype in the form of class for all objects of same kind.

32.3.Function Templates

A function template can be parameterized to operate on different types of data types.

Declaration:

We write template keyword above any function make any function as template function, they can be declared in any one of the following ways,

```
template< class T >
void funName( T x );

// OR

template< typename T >
void funName( T x );

// OR

template< class T, class U, ... >
void funName( T x, U y, ... );
```

Note here T is typename of class name and we use this T instead of data type(s) for which we want our function to work as template.

For Example – Function Templates
Following function template prints an array having almost any type of elements (note the use of T instead of int or float or char in implementation of function):

```
template< typename T >
void printArray( T* array, int size )
{
        for ( int i = 0; i < size; i++ )
                cout << array[ i ] << ", "; // here data type of array is T
}
```

Similarly we can also write same function as,

```
template< class T >
void printArray( T* array, int size )
{
        for ( int i = 0; i < size; i++ )
                cout << array[ i ] << ", "; // here data type of array is T
}
```

There is no difference in above two implementations.

 © Virtual University of Pakistan

Template function will be instantiated for a particular data type according to passed argument as shown below,

```
int main() {

        int iArray[5] = { 1, 2, 3, 4, 5 };
        void printArray( iArray, 5 );  // Instantiated for int[] as passed array is of type int []

        char cArray[3] = { 'a', 'b', 'c' };
        void printArray( cArray, 3 ); // Instantiated for char[] as argument is of type char []
        return 0;
}
```

Explicit Type Parameterization:

In case a function template does not have any parameter then we have to explicitly mention the data type for which we want to create that function as shown below,

```
        template <typename T>
        T getInput() {
                T x;
                cin >> x;
                return x;
        }
```

Explicit Type Parameterization

```
int main() {
        int x;
        x = getInput();          // Error!

        double y;
        y = getInput();          // Error!
        return 0;
}
```

Explicit Type Parameterization

```
int main() {
        int x;
        x = getInput< int >();

        double y;
        y = getInput< double >();
        return 0;
}
```

User-defined Specializations

A template compiler generated code may not handle all the types successfully; in that case we can give explicit specializations for a particular data type(s).

For example suppose we have written a function isEqual(…, …) that compares two values of data type and return true or false depending upon the values are equal or not,

```
template< typename T >
bool isEqual( T x, T y ) {
        return ( x == y );
}
```

isEqual (6,6) should return true
isEqual (6,7) should return false
isEqual (6.6,6.6) should return true
isEqual (6.5,6.6) should return false
isEqual ('A','A') should return true
isEqual ('A','a') should return false

Until here the function will work correctly but consider the statement below,

isEqual ("abc","xyz")

This is instantiation of isEqual function for built in type char [] or char *[14], this function will fail to give correct result simply because we have given its implementation as

```
return (x == y);
```

So here it will be translated by compiler in,

```
return ( char *  == char *);  or  return ( char []  == char []);
```

As arrays consists of many elements, comparison of arrays in this way is not possible it will simply compare first element of both arrays and will return result, this is given in code below,

```
#include <cstdlib>
#include <iostream>

using namespace std;

template< typename T >
bool isEqual( T x, T y ) {
        return ( x == y );
}
```

[14] In C++, both char [] and char * are of same data types.

 © Virtual University of Pakistan

```
int main (){
        cout<<isEqual( 5, 6 ); // Ok
        cout<<isEqual( 7.5, 7.5 );      // Ok
        cout<<isEqual( "abc", "bca" ); // Logical Error!

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

So all statements with starting same characters like below will return **true**,

isEqual("**a**bc", "**a**cc");
isEqual("**b**adac", "**b**acc");
isEqual("**c**afaa", "**c**cda");

This is logical error, solution of this problem is that we give our own correct implementation of **isEqual** template function for char * data type, for this we can write specialization code below general template function as follows,

Example – User Specializations

```
#include <cstdlib>
#include <iostream>

using namespace std;

template< typename T >
bool isEqual( T x, T y ) {
        return ( x == y );
}

template< >
bool isEqual< const char* >(
        const char* x, const char* y ) {
        return ( strcmp( x, y ) == 0 );
}


int main (){
        cout<<isEqual( 5, 6 );  // OK
        cout<<isEqual( 7.5, 7.5 );        // OK
        cout<<isEqual( "abc", "aba" ); //OK will return False


   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Lecture No.33

Recap

- Templates are generic abstractions
- C++ templates are of two kinds
  - Function Templates
  - Class Templates
- A general template can be **specialized** to specifically handle a particular type like we did for char []

33.1. Multiple Type Arguments

Suppose we want to write code to convert different types into one another (like char to int or int to char or float to int or int to float), the problem is same we have to write many functions corresponding to each type, as no of types will increase the required no. of functions will also increase, the concept of templates can be used here as well to write general function to convert one type into another, in this case we will need two type arguments as shown below,

imp for mcq

```
template< typename T, typename U >
T my_cast( U u ) {
        return (T)u; // U type will be converted to T type and will be returned
}

int main() {
        double d = 10.5674;
        int j = my_cast( d );    //Error
        int i = my_cast< int >( d );

   // need to explicity mention about type of T (int in this case) as it is used only
for
  // return type not as parameter

        return 0;
}
```

33.2. User-Defined Types

Besides primitive types, user-defined types can also be passed as type arguments to templates, compiler performs static type checking to diagnose type errors.
Consider the String class without overloaded operator "=="

```
class String {
        char* pStr;
        …
        // Operator "==" not defined
};

template< typename T >
bool isEqual( T x, T y ) {
```


© Virtual University of Pakistan

```
        return ( x == y );
}
int main() {
        String s1 = "xyz", s2 = "xyz";
        isEqual( s1, s2 );      // Error!
        return 0;
}
```

We can use String class objects as arguments to template function **isEqual** also but in this case we should have defined overloaded == operator for our string class as friend function because this operator is being used in **isEqual** function,

```
class String {
        char* pStr;
        …
        friend bool operator ==(    const String&, const String& );
        };

bool operator ==( const String& x, const String& y ) {
        return strcmp(x.pStr, y.pStr) == 0;
}

template< typename T >
bool isEqual( T x, T y ) {
        return ( x == y );
}

int main() {
        String s1 = "xyz", s2 = "xyz";
        isEqual( s1, s2 );      // OK
        return 0;
}
```

33.3.Overloading vs. Templates

Function templates are used when we want to have exactly identical operations on different data types in case of function templates we can not change implementation from data type to data type however we can specialize implementation for a particular data type.
In case we want to have similar operations on different data types we need function overloading. In case of function overloading, we can give similar but slightly different implementation for each data type.

Example: Overloading vs. Templates

- '+' operator is **overloaded** for different types of opeerands (different implementation in each case).
- A single function template can calculate **sum** of array of many types.

Function Overloading

```
String operator +( const String& x,          const String& y ) {
        String tmp;
        tmp.pStr = new char[strlen(x.pStr) + strlen(y.pStr) + 1 ];
        strcpy( tmp.pStr, x.pStr );
        strcat( tmp.pStr, y.pStr );
        return tmp;
}

String operator +( const char * str1,  const String& y ) {
        String tmp;
        tmp.pStr = new char[ strlen(str1) + strlen(y.pStr) + 1 ];
        strcpy( tmp.pStr, str1 );
        strcat( tmp.pStr, y.pStr );
        return tmp;
}
```

Templates

```
template< class T >
T sum( T* array, int size ) {
        T sum = 0;

        for (int i = 0; i < size; i++)
                sum = sum + array[i];

        return sum;
}
```

33.4. Template Arguments as Policy:

We can change behaviour of a template using template parameter. We can pass a template argument to enforce some rule (policy). For example see the problem statement below:

"Write a function that compares two given character strings."

This function is similar to built in string comparison function **strcmp** with the difference that it can perform both case sensitive and case insensitive comparisons.

© Virtual University of Pakistan

33.5.First Solution:

We can write two separate functions for both cases and call any of them as required,

```
int caseSencompare( char* str1, char* str2 )
{
        for (int i = 0; i < strlen( str1 ) && i < strlen( str2 ); ++i)
                if ( str1[i] != str2[i] )
                        return str1[i] - str2[i];

        return strlen(str1) - strlen(str2);
}
```

This function will return 0 in case string length of both strings is same and they have identical elements otherwise it will return 1.

```
int nonCaseSencompare( char* str1,  char* str2 )
{
        for (int i = 0; i < strlen( str1 ) && i < strlen( str2 ); i++)
                if ( toupper( str1[i] ) != toupper( str2[i] ) )
                        return str1[i] - str2[i];

        return strlen(str1) - strlen(str2);
}
```

This function will return 0 in case string length of both strings is same and they have same alphabets (ignoring they are in lower case or in upper case) otherwise it will return 1.

33.6.Second Solution:

We write a single compare function and pass a **bool** type parameter to indicate type of comparison and from this function we return result based on passed **bool** parameter used to indicate case sensitive or case insensitive comparison,

```
int compare( char* str1, char* str2,  bool caseSen )
{
        for (int i = 0; i < strlen( str1 ) && i < strlen( str2 ); i++)
                if ( … )
                        return str1[i] - str2[i];

        return strlen(str1) - strlen(str2);
}

// if condition:
```

(caseSen && str1[i] != str2[i]) || (!caseSen && toupper(str1[i]) != toupper(str2[i]))
Here bool variable caseSen will work as flag, and will activate either left sub-expression or right sub-expression in **if-statement**, in case we are performaing case sensitive comparison caseSen will be true (1) and will activate sub-expression,
(caseSen && str1[i] != str2[i])
and in case caseSen is false it will activate second sub-expression involving **!caseSen** as given below,
(!caseSen && toupper(str1[i]) != toupper(str2[i])
other logic is same as in case 1 functions.

33.7. Third Solution

Third solution is most elegant solution out of all these, in which we write two classes one for case sensitive and other for case insensitive comparison, and pass one of these classes as argument when instantiating template compare function, in compare function **isEqual** function of passed class is being called to perform either case sensitive or case insensitive comparison,

```
class CaseSenCmp {
public:
        static int isEqual( char x, char y ) {
                return x == y;
        }
};

class NonCaseSenCmp {
public:
        static int isEqual( char x, char y ) {
                return toupper(x) == toupper(y);
        }
};

template< typename C >
int compare( char* str1, char* str2 )
{
        for (int i = 0; i < strlen( str1 ) && i < strlen( str2 ); i++)
                if ( !C::isEqual (str1[i], str2[i]) )
                        return str1[i] - str2[i];

        return strlen(str1) - strlen(str2);
};

int main() {
        int i, j;
        char *x = "hello", *y = "HELLO";
        i = compare< CaseSenCmp >(x, y);
        j = compare< NonCaseSenCmp >(x, y);
        cout << "Case Sensitive: " << i;
        cout << "\nNon-Case Sensitive: “<< j << endl;
        return 0;
```

 © Virtual University of Pakistan

```
}
```

Sample Output

```
Case Sensitive: 32      // Not Equal
Non-case Sensitive: 0// Equal
```

33.8. Default Policy

We can set default class type as default comparison type as we set default parameters in case of constructors,

```
template< typename C = CaseSenCmp >
int compare( char* str1, char* str2 )
{
        for (int i = 0; i < strlen( str1 ) && i < strlen( str2 ); i++)
                if ( !C::isEqual
                                        (str1[i], str2[i]) )
                        return str1[i] - str2[i];

        return strlen(str1) - strlen(str2);
};

int main() {
        int i, j;
        char *x = "hello", *y = "HELLO";
        i = compare(x, y);
        j = compare< NonCaseSenCmp >(x, y);
        cout << "Case Sensitive: " << i;
        cout << "\nNon-Case Sensitive: “
                        << j << endl;
        return 0;
}
```

Lecture No.34
Generic Algorithms - A Case Study

Consider the template **printArray** function we wrote in previous lecture,

Printing an Array

```
template< typename T >
void printArray( T* array, int size )
{
        for ( int i = 0; i < size; i++ )
                cout << array[ i ] << ", ";
}
```

Although this function will work for all data types but still it is taking array as an argument so this function still depends upon nature of data structure used to pass data, we want this function to be independent from data structure as well means it should print single values as well as arrays of basic data types.

34.1. Generic Algorithms

We want to provide such implementation that is independent of data structures also for example in case of printing of values we want printing of both single values and arrays, this can be achieved using *Generic Programming* in which a function work for all types of containers, let us see how we can make a function generic step by step with the help of an example, consider the **find** function below it is similar to **printArray** function with the difference that it tries to find an element in an array and return pointer to that element if it is found in the array otherwise it returns zero (NULL).

Find function that tries to find an integer value in an integer array

```
const int* find( const int* array, int _size, int x ) {
        const int* p = array;
        for (int i = 0; i < _size; i++) {
                if ( *p == x )
                        return p;
                p++;
        }
        return 0;
}
```

First step obviously will be to write it as template so that it may work for all data types instead of only integers,
We write it as template function,

Template function to find a value within an array

```
template< typename T >
T* find( T* array,int _size, const T& x ) {
```


© Virtual University of Pakistan

```
        T* p = array;
        for (int i = 0; i < _size; i++) {
                if ( *p == x )
                        return p;
                p++;
        }
        return 0;
}
```

In next step we simply pass a pointer that is pointing one location next after array instead of passing size of array as shown below, (this will make our code simplified)

```
template< typename T >
T* find( T* array, T* beyond,  const T& x ) {
        T* p = array;
        while ( p != beyond ) {
                if ( *p == x )
                        return p;
                p++;
        }
        return 0;
}
```

We also change the return statement of the function by returning the beyond pointer instead of zero in case element is not found, so now we will not check the return value for NULL pointer we will simply check whether it points to any value or it points to beyond pointer, (we are doing it so that we can use a single return statement to return a single pointer as we have done below)

```
template< typename T >
T* find( T* array, T* beyond, const T& x ) {
        T* p = array;
        while ( p != beyond ) {
                if ( *p == x )
                        return p;
                p++;
        }
        return beyond;
}
```

using single return statement,

```
template< typename T >
```

```
T* find( T* array, T* beyond,  const T& x ) {
        T* p = array;
        while ( p != beyond && *p != x )
                p++;
        return p;
}
```

Now our code is so generic that it will work for all types of containers (data structures) supporting two operations,

- Increment operator (++) as we are incrementing value in this container
- Dereference operator (*) as we are getting value from container for comparison by dereferencing

Although this template function will now work for all containers in same way but it has one limitation that we will need to pass container pointers in this function this is against the concept of generic programming so we simply remove pointer notation from our code,

```
template< typename P, typename T >
P find( P start, P beyond, const T& x ) {
        while ( start != beyond && *start != x )
                        start++;
        return start;
}
```

Now this implementation will work for all references of containers as shown in code below,

```
int main() {
        int iArray[5];
        iArray[0] = 15;
        iArray[1] = 7;
        iArray[2] = 987;
        …
        int* found;
        found = find(iArray, iArray + 5, 7);
        return 0;
}
```

We can apply the same concept of Generic Algorithms to class templates and develop a generic Vector class that will work for all built-in types as shown below,

34.2.Class Templates

© Virtual University of Pakistan

As we saw before a single class template provides functionality to operate on different types of data and in this way facilitates reuse of classes (like we wrote our own array class to overcome the limitations of built-in array class in c++).

We can definition a class template as follows:

- template< class T > class Xyz { … };
- template< typename T > class Xyz { … };

Now we write a template Vector class using the concept of Generic Algorithms such that it can store everything, basically it will act as container to store anything in it, it can store objects that are them self collections like arrays or it can store basic data types. This will be possible because we are going to implement this Vector class using the concept of Generic Algorithms.

34.3.Example – Class Template

A **Vector** class template can store data elements of different types, without templates, we need a separate **Vector** class for each data type.

We start with the basic definition of Vector class using templates,

Class Template

```
template< class T >
class Vector {
private:
        int size;
        T* ptr;
public:
        Vector<T>( int = 10 );
        Vector<T>( const Vector< T >& );
        ~Vector<T>();
        int getSize() const;
        const Vector< T >& operator =(
                                const Vector< T >& );
        T& operator []( int );
};
```

Its implementation is,

```
template< class T >
Vector<T>::Vector<T>( int s ) {
        size = s;
        if ( size != 0 )
                ptr = new T[size];
        else
```

```
                ptr = 0;
}
template< class T >
Vector<T>:: Vector<T>(
                        const Vector<T>& copy ) {
        size = copy.getSize();
        if (size != 0) {
                ptr = new T[size];
                for (int i = 0; i < size; i++)
                        ptr[i] = copy.ptr[i];
        }
        else ptr = 0;
}

template< class T >
Vector<T>::~Vector<T>() {
        delete [] ptr;
}

template< class T >
int Vector<T>::getSize() const {
        return size;
}
template< class T >
const Vector<T>& Vector<T>::operator
                        =( const Vector<T>& right) {
        if ( this != &right ) {
                delete [] ptr;
                size = right.size;

                if ( size != 0 ) {
                        ptr = new T[size];
                        for(int i = 0; i < size;i++)
                                ptr[i] = right.ptr[i];
                }
                else
                        ptr = 0;
        }
        return *this;
}
template< class T >
T& Vector< T >::operator [](        int index ) {
        if ( index < 0 || index >= size ) {
                cout << "Error: index out of
                range\n";
                exit( 1 );
```

```
        }
        return ptr[index];
}
```

We can create this vector class instances for int or char data type as given below,

imp for mcq & short
Write syntax for int type
nad char type

Vector< int > intVector;
Vector< char > charVector;

This Vector class is parameterized class and will always be instantiated for a particular type only. Now we can not create object of type Vector only it will be instantiated for a particular data type like Vector<int> or Vector <float> and so on…

Lecture No.35

35.1.Member Templates:

Member functions of a template class implicitly become functions templates; they work for instantiations (int, char, float, double so on…) of that class, however there are some situations where we need explicit template functions in our class taking more template parameters other than one implicit template parameter (parameter given to this class as parameter while creating its object).
A class or class template can have member functions that are themselves templates

```
template<typename T> class Complex {
        T real, imag;
public:
        // Complex<T>( T r, T im )
        Complex( T r, T im ) :
                real(r), imag(im) {}
        // Complex<T>(const Complex<T>& c)
Complex(const Complex<T>& c) :
                real( c.real ), imag( c.imag ) {}
        …
};
```

Note that in C++ while declaring class there is no need to mention template parameter for class member functions as compiler implicitly understand it, however if we are using some other template parameter like we are doing for copy constructor then we need to give its name as well.
Now see that main function for this class in which we are assigning Complex class float instance to double, it will result in an error,

```
int main() {
        Complex< float > fc( 0, 0 );
        Complex< double > dc = fc; // Error
        return 0;
}
```

Because, our Complex copy constructor is taking argument of same template type T so this copy constructor will work for statements like,

Complex< float > f1c(0, 0);
Complex< float > f2c = f1c;

Complex< double > d1c(0, 0);
Complex< double > d2c = d1c;

But will not work for statements like,

Complex< float > fc(0, 0);

© Virtual University of Pakistan

```
Complex< double > dc = fc;

Complex< double > d1c( 0, 0 );
Complex< float > d2c = d1c;
```

When we created Complex object for double compiler generated class for double like shown below,

```
class Complex<double> {
        double real, imag;
public:
        Complex( double r, double im ) :real(r), imag(im)
        {}
        Complex(const Complex<double>& c) :real( c.real ), imag( c.imag )
        {}
        …
};
```

So, there is need of some sort of overloading of copy constructor such that we can assign two instances of Complex class for two different data types.

Now we change copy constructor as function template explicitly, so that it may work for copy of different types of data types as well,

```
template<typename T> class Complex {
        T real, imag;
public:
        Complex( T r, T im ) :
                real(r), imag(im) {}
        template <typename U>
// this copy constructor is now taking two template parameters one implicit T
// and other explicit U
        Complex(const Complex<U>& c) :
                real( c.real ), imag( c.imag ) {}
        …
};
```

Now assignment of float Complex instance to double instance will fine,

```
int main() {
        Complex< float > fc( 0, 0 );
        Complex< double > dc = fc; // OK
        return 0;
}
```

Because, here copy constructor
Complex(const Complex<U>& c) :

Will be instantiated for implicit template type (T) double and explicit template type (U) float because now our Copy constructor is acting as explicity template function written for two different parameters.

Important points to Note:

1. Only that instantiation of copy constructor is generated that is required.
2. Good compilers only generate required template function instances for Complex class instantiation for a particular data type as shown below,

For statement: **Complex< double > dc = fc;**
the functions that will be generated for float and double instance of complex class are shown below,

<double> Instantiation

```
class Complex<double> {
        double real, imag;
public:
        Complex( double r, double im ) :
                real(r), imag(im) {}
        template <typename U>
        Complex(const Complex<U>& c) :
                real( c.real ), imag( c.imag ) {}
        …
};
```

<float> Instantiation

```
class Complex<float> {
        float real, imag;
public:
        Complex( float r, float im ) :
                real(r), imag(im) {}
// No Copy Constructor code is generated as there is no need for it
        …
};
```

This approach avoid code bloat (unnecessary code generation).

35.2. Class Template Specialization

Like function templates a class template may also not handle all the types successfully, for example for char arrays (char *) the behaviour of template class vector may be not be as desired as shown below,

Vector class to store integers

```
int main() {
        Vector< int > iv1( 2 );
        iv1[0] = 15;
```

 © Virtual University of Pakistan

```
        iv1[1] = 27;
        Vector< int > iv2( iv1 );
        Vector< int > iv3( 2 );
        iv3 = iv1;
        return 0;
}
```

Vector class to store char arrays (char *)

```
int main() {
        Vector< char* > sv1( 2 );
        sv1[0] = "Aamir";
// compiler will generate a const C String having value Aamir and will assign its
//pointer to sv1[0]
        sv1[1] = "Nasir";
        Vector< char* > sv2( sv1 ); // issue of shallow copy
        Vector< char* > sv3( 2 );
        sv3 = sv1; // issue of shallow copy
        return 0;
}
```

We can write explicit specialization for Vector class for char arrays as we wrote for template function isEqual,

```
template<>
class Vector< char* > {
private:
        int size;
        char** ptr;
public:
        // Vector< char* >( int = 10 );
        Vector( int = 10 );
        Vector( const Vector< char* >& );
        virtual ~Vector();
        int getSize() const;
        const Vector< char* >& operator =( const Vector< char* >& );
        const char*& operator []( int );
        void insert( char*, int );
};
template<>
Vector<char*>::Vector(int s) {
        size = s;
        if ( size != 0 ) {
                ptr = new char*[size];
                for (int i = 0; i < size; i++)  ptr[i] = 0;
        }
        else
                ptr = 0;
}
template<>
```

```
Vector< char* >::Vector( const Vector<char*>& copy ) {
        size = copy.getSize();
        if ( size == 0 ) {
                ptr = 0;
                return;
        }
        ptr = new char*[size];
        for (int i = 0; i < size; i++)
                if ( copy.ptr[i] != 0 ) {
                        ptr[i] = new char[ strlen( copy.ptr[i] ) + 1 ];
                        strcpy(ptr[i], copy.ptr[i]);
                }
                else
                        ptr[i] = 0;
}

template<>
Vector<char*>::~Vector() {
        for (int i = 0; i < size; i++)
                delete [] ptr[i];

        delete [] ptr;
}

template<>
int Vector<char*>::getSize() const {
        return size;
}

template<>
const Vector<char*>& Vector<char*>::
operator=(const Vector<char*>& right)
{
        if ( this == &right )
                return *this;
        for (int i = 0; i < size; i++)
                delete [] ptr[i];
        delete [] ptr;
size = right.size;
if ( size == 0 ) {
        ptr = 0;
        return *this;
}
ptr = new char*[size];
for (int i = 0; i < size; i++)
        if ( right.ptr[i] != 0 ) {
                ptr[i] = new char[strlen( right.ptr[i] ) + 1];
                strcpy( ptr[i], right.ptr[i] );
        }
        else
                ptr[i] = 0;
}
```

 © Virtual University of Pakistan

```
template<>
const char*& Vector<char*>:: operator []( int index ) {
        if ( index < 0 || index >= size ) {
                cout << "Error: index out of range\n";
                exit( 1 );
        }
        return ptr[index];
}

template<>
void Vector< char* >::insert( char* str, int i ) {
        delete [] ptr[i];
        if ( str != 0 ) {
                ptr[i] = new char[strlen(str)+1];
                strcpy( ptr[i], str );
        }
        else
                ptr[i] = 0;
}
```

This code is similar to general template Vector class with the difference that now we are allocating and taking care of dynamic memory associated with char arrays.
The main program showing usage of above main program is given below,

```
int main() {
        Vector< char* > sv1( 2 );
        sv1[0] = "Aamir";
   // Error as now we have to changed code of overloaded subscript operator to return //const Vector pointer now and we can not assign new value to constant reference //of Vector now we have to use insert function explicity written for that purpose
        sv1.insert( "Aamir", 0 );
        sv1.insert( "Nasir", 1 );
        Vector< char* > sv2( sv1 );
        Vector< char* > sv3( 2 );
        sv3 = sv1;
        return 0;
}
```

Lecture No.36

Recap

We saw in previous lecture that how we can implement our programming problems easily using Generic Algorithms. Then we saw how we can add explicit template functions to our Class templates to add our desired functionality.
A class template may not handle all the types successfully explicit specializations are required to deal such types we can implement concept of template specialization for such classes as well as we did for function templates.

Now, we move forward to see other cases for member templates,

36.1.Member Templates Revisited

We can add member templates for ordinary classes as well, for example following code is adding any instance of Complex class to ComplexSet class, (as we know complex class can be instantiated for int, float or double and ComplexSet class will be collection of Complex class instantiations) as shown below,

Complex Class objects,

real	8
img	3

Complex<int>

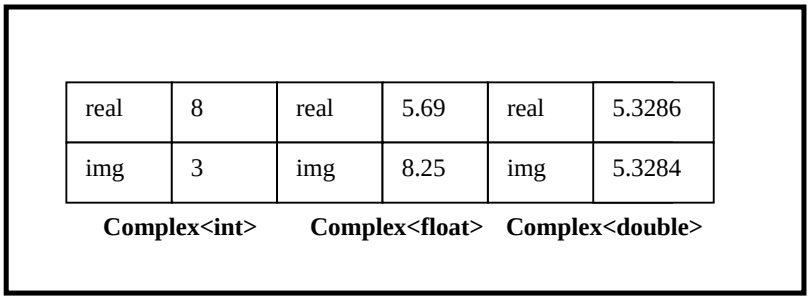
real	5.69
img	8.25

Complex<float>

real	5.3286
img	5.3284

Complex<double>

© Virtual University of Pakistan



ComplexSet Object

```
class ComplexSet {

        template< class T >
        insert( Complex< T > c )  // any instance of complex class
                        {
// Add instance Complex class to Complex set class having many // Complex class
instances
}
        };
int main() {
        Complex< int > ic( 10, 5 );
        Complex< float > fc( 10.5, 5.7 );
        Complex< double > dc( 9.567898, 5 );
        ComplexSet cs;
        cs.insert( ic );
        cs.insert( fc );
        cs.insert( dc );
        return 0;
}
```

36.2. Partial Specialization:

We can also perform partial specialization instead of complete specialization, a partial specialization of a template exists between general specialization and complete specialization for example we can specialize a class to behave in a certain manner in case of pointers or in case of parameter of a certain type, see the example below to understand the difference between complete and partial specialization,

most imp for mcq 304 & 304p

```
template< class T, class U, class V > // general template
template< class T, class U, int > // partial specialization
template< class T, float , int  > // partial specialization
template< int , float , int  > // complete specialization
```

In partial specialization, the number of template parameters remains the same, however, their nature varies (they become more specific).

Example – Partial Specialization

```
template< class T >
class Vector { };
template< class T >
class Vector< T* > { };   // Here T can take any type  pointer
```

Example – Partial Specialization

```
template< class T, class U, class V >
class A {};
template< class T, class V >
class A< T, T*, V > {};  // the parameters in header of are two but in class we are using same three parameters
template< class T, class U, int I >
class A< T, U, I > {};  // here we have changed third parameter to non type parameter[15]
template< class T >
class A< int, T*, 5 > {};   // here we have changed first parameter to non type parameter and second is T * any data type pointer and third one is constant expression 5
```

Example – Complete Specialization

```
template< class T >
class Vector { };

template< >
class Vector< char* > { };
```

Example – Complete Specialization

```
template< class T, class U, class V >
class A {};

template< >
class A< int, char*, double > {};
```

36.3.Function Templates

Similar to class templates a function template may also have partial specializations,

[15] non type parameters are those parameters which are not template parameters

 © Virtual University of Pakistan

Example – Partial Specialization

```
template< class T, class U, class V >
void func( T, U, V );

template< class T, class V >
void func( T, T*, V );

template< class T, class U, int I >
void func( T, U );

template< class T >
void func( int, T, 7 );
```

36.4.Complete Specialization

We have already used this complete specialization in case of function templates,

```
template< >
        bool isEqual< const char* >(
                const char* x, const char* y ) {
                return ( strcmp( x, y ) == 0 );
        }
```

Example

Consider the following template

```
template< typename T >
        bool isEqual( T x, T y ) {
                return ( x == y );
        }
```

Following partial specialization of this function deals with pointers to objects,

```
template< typename T >
        bool isEqual( T* x, T* y ) {
                return ( *x == *y );
        }
```

So complete having all types of specializations for isEqual function is shown below,

```
template< typename T >
        bool isEqual( T x, T y ) {
                return ( x == y );
        }
template< typename T >
        bool isEqual( T* x, T* y ) {
```

```
                return ( *x == *y );
        }
template< >
        bool isEqual< const char* >(
                const char* x, const char* y ) {
                return ( strcmp( x, y ) == 0 );
        }
```

36.5. Using Different Specializations

The code below shows how we can use different type of specializations for isEqual function given above,

```
int main() {
        int i, j;
        char* a, b;
        Shape *s1 = new Line();
        Shape *s2 = new Circle();
        isEqual( i, j );   // Template
        isEqual( a, b );   // Complete Sp.
        isEqual( s1, s2 ); // Partial Sp.
        return 0;
}
```

36.6. Non-type Parameters

The parameters given in template definition other than those used for mentioning templates types are called non type parameters, for example,

mcqs 304p

template <class T, class U, **int I>**

03/02/2025

Here int I is not type parameter.

Template parameters may include non-type parameters, the non-type parameters may have default values, for example,

template <class T, class U, **int I = 5>**

They are treated as constants and are commonly used is static memory allocation mean when we want to pass the length of memory we need in template at compile time (statically) the example below shows all this,

36.7. Example – template class Array

© Virtual University of Pakistan

Consider the Array class again that we discussed in template classes introduction, in this template array class we are creating a c++ array of any built in data type by passing array size in Array object constructor as shown below,

```
template< class T >
class Array {
private:
        T* ptr;
public:
        Array( int size );
        ~Array();
        …
};
template< class T >
Array<T>::Array() {
        if (size > 0)
                ptr = new T[size];
        else
                ptr = NULL;
}
int main() {
        Array< char > cArray( 10 );
        Array< int > iArray( 15 );
        Array< double > dArray( 20 );
        return 0;
}
```

We can do the same by passing the array size as non type parameter while creating an Array class object itself, for this we will need to change to template class definition by adding a non type parameter SIZE as shown below,

```
template< class T, int SIZE >
class Array {
private:
        T ptr[SIZE];
public:
        Array();
        …
};
```

Now we can simply pass array size as argument while creating Array class object,

```
int main() {
        Array< char, 10 > cArray;
        Array< int, 15 > iArray;
        Array< double, 20 > dArray;
```

```
        return 0;
}
```

36.8. Default Non-type Parameters

We can set default value for this non type parameters, as we do for parameters passed in ordinary functions,

```
template< class T, int SIZE = 10 >
class Array {
private:
        T ptr[SIZE];
public:
        void doSomething();
        …
};
```

Now default value will be used if we will not pass any value for array size,

```
int main() {
        Array< char > cArray; // here Array of size 10 will be created
        return 0;
}
```

36.9. Default Type Parameters

We can also specify default type for type parameters (template parameters like T, U, V), consider the Vector class again we can mention default type of Vector class (that type will be used in case we have not mentioned any type while creating Vector class object)

```
        template< class T = int >  // default type for Vector class is now int
        class Vector {
                …
        };

        Vector< > v;  // same as Vector< int > v
```

Lecture No.37

© Virtual University of Pakistan

Lecture No. 37

37.1.Resolution Order

Consider the following specializations of template **Vector** class,

```
template< typename T >
class Vector { … };

template< typename T >
class Vector< T* > { … };

template< >
class Vector< char* > { … };
```

As these all are template implementations of class **Vector,** when compiler have to use this template class having many specializations from partial to complete, compiler searches all these specializations in a particular order called **resolution order**, so **resolution order** it is the sequence in which compiler searches for required template specialization and is given below,

short question:
304 ma aya tha
31/01/2025

a. First of all compiler looks for complete specialization
b. If it can not find any required complete specialization then it searches for some partial specialization
c. In the end it searches for some general template

So in other words we can say that compiler searches template specializations from more specific to more general.

Example – Resolution Order

The code below shows which particular instantiation of template **Vector** class will be used,

```
int main() {
        Vector< char* > strVector;
                // Vector< char* > instantiated (complete specialization used)

        Vector< int* > iPtrVector;
                // Vector< T* > instantiated (partial specialization used)

        Vector< int > intVector;
                // Vector< T > instantiated (general specialization used)
        return 0;
}
```

Explanation of code:

In all three cases above, compiler will search for required template specialization in the order given below,

1. Complete specialization (Vector< char* >)
2. Partial specialization (Vector< T* >)
3. General specialization (Vector< T>)

- When compiler will find the following line,

Vector< char* > strVector;

It will find corresponding complete specialization and will stop there.

- When compiler will find the following line,

Vector< int* > iPtrVector;

It will start its searching from complete specializations available but there is only one complete specialization for char * (**Vector< char* >**) and there is no complete specialization for int *, so compiler will move to partial specialization and will try to match some with int *, compiler will find one **(Vector <T** *>) that is working for all pointer data types and therefore it will use that.

- When compiler will find the following line,

Vector< int > intVector;

It will start from complete specializations available but there is only one complete specialization for char * (**Vector< char* >**) and there is no complete specialization for int, so compiler will move to partial specialization and will try to match some with int, compiler will find one partial speciation **(Vector <T *>)** that is working for all pointer data types only hence not applicable here, then compiler will move to general template class Vector <T> and will use it.

37.2.Function Template Overloading

We can specialize function templates also, this is called function template overloading,
Consider the specializations of template function sort,

```
template< typename T >
void sort( T );  // general template function
template< typename T >
void sort( Vector< T > & );  // specialization for template Vector class instantiated for any data type
template< >
void sort< Vector<char*> >(Vector< char* > & );  // specialization for template Vector instantiated for char * data type
void sort( char* ); // specialization for char * data type.
```


© Virtual University of Pakistan

37.3.Resolution Order

Compiler searches target of a function call in the following order in case of function of templates,

- Ordinary Function
- Complete Specialization
- Partial Specialization
- Generic Template

This is the same order as was in case of class template with the addition of ordinary functions (non template functions with the same name).

Example – Resolution Order

```
int main() {

        char* str = "Hello World!";
        sort(str);  //  Ordinary function sort( char* )

        Vector<char*> v1 = {"ab", "cd", … };
        sort(v1);  //Complete specialization sort( Vector<char*> & )


        Vector<int> v2 = { 5, 10, 15, 20 };
        sort(v2); // partial specialization sort( Vector<T> &)

        int iArray[] = { 5, 2, 6 , 70 };
        sort(iArray); // general template sort( T )

        return 0;
}
```

37.4.Templates and Inheritance

We can use inheritance comfortably with templates or their specializations, but we must follow one rule:

- If we have a template class then all classes derived from it should also be class templates.
- Derived class must take at least as many template parameters as the base class requires for an instantiation.

37.5.Derivations in case of a General Template class

- A class template may inherit from another class template. Consider the following code segment in which we are defining a template class A and we are deriving another class B from it which is also a template class.

```
        template< class T >
```

```
        class A
        { … };

        template< class T >
        class B : public A< T >
// same template parameter T in both classes declarations ensures that both classes will be instantiated for same types
        { … };

int main() {
        A< int > obj1;
        B< int > obj2;
        return 0;
}
```

- A partial specialization may inherit from a class template

```
        template< class T >
        class B< T* > : public A< T >// same template parameter T is used here also to //ensure that both classes will be instantiated for same data types
        { … };

int main() {
        A< int > obj1;
        B< int* > obj2;
        return 0;
}
```

- Complete specialization or ordinary class cannot inherit from a class template because as you can see complete specialization is in fact instance of the class for one data type and in this way it is similar to ordinary class as ordinary class also can be instantiated for its members of a particular data type,

```
template< >
class B< char* > : public A< T >
        { … };
// Error: 'T' undefined, derived class is taking less parameters than base class

class B : public A< T >
        { … };
// Error: 'T' undefined, derived class is taking less parameters than base class
```

Derivations in case of a partially specialized class

- A class template may inherit from a partial specialization

```
template< class T >
```


© Virtual University of Pakistan

```
        class A
        { … };

        template< class T >
        class A< T* >
        { … };
template< class T >
class B : public A< T* >
{ … } ;

int main() {
        A< int* > obj1;
        B< int > obj2;
        return 0;
}
```

- A partial specialization may inherit from a partial specialization

```
template< class T >
        class B< T* > : public A< T* >
        { … };

int main() {
        A< int* > obj1;
        B< int* > obj2;
        return 0;
}
```

- Complete specialization or ordinary class cannot inherit from a partial specialization

```
        template< >
        class B< int* > : public A< T* >
        { … };  // Error: Undefined 'T'

        class B : public A< T* >
        { … };  // Error: Undefined 'T'
```

Derivations in case of Completely Specialized class

```
template< class T >
class B : public A< float* >
{ … };
```

```
int main() {
        A< float* > obj1;
        B< int > obj2;
        return 0;
}
```

- A partial specialization may inherit from a complete specialization

```
template< class T >
        class B< T* > : public A< float* >
        { … };

int main() {
        A< float* > obj1;
        B< int* > obj2;
        return 0;
}
```

- A complete specialization may inherit from a complete specialization

```
template< >
        class B< double* > : public A< float* >
        { … };

int main() {
        A< float* > obj1;
        B< double* > obj2;
        return 0;
}
```

- An ordinary class may inherit from a complete specialization

```
class B : public A< float* >
        { … };

int main() {
        A< float* > obj1;
        B obj2;
        return 0;
}
```

Derivations in case of Ordinary Class

- A class template may inherit from an ordinary class

```
class A
```


© Virtual University of Pakistan

```
        { … };

        template< class T >
        class B : public A
        { … };
int main() {
        A obj1;
        B< int > obj2;
        return 0;
}
```

- A partial specialization may inherit from an ordinary class

```
class A{
   };

 template<class T>
 class B {};
template <class T>
class B<T*>: public A{};


int main() {
   A  obj1;
  B <int *> obj2;
    return 0;
}
```

- A complete specialization may inherit from an ordinary class

```
        template <class T>
         class B{};
         template< >
        class B< char* > : public A
        { … };

int main() {
        A obj1;
        B< char* > obj2;
        return 0;
}
```

Lecture No.38

38.1.Templates and Friends

Templates or their specializations are also compatible with friendship feature of C++ as they are with inheritance.
We consider the following rules when dealing with templates and friend functions and classes.

38.2.Templates and Friends – Rule 1

When an ordinary function or class is declared as friend of a class template then it becomes friend of each instantiation of that template class.
Consider the code below in which we are declaring a class A as friend of a class B and in this class A, we are accessing private data of class B without any error,
In case of friend classes,

```
class A {
        …
};

template< class T >
class B {
        int data;
        friend A; // declaring A as friend of B
        …
};

class A {
        void method() {
                B< int > ib;
                B< char > cb
                ib.data = 5; // OK: Accessing private member 'data' for class B instantiation ib
                cb.data = 6; // OK: Accessing private member 'data' for class B instantiation cb
        }
};
```

In case of friend functions,

```
void doSomething( B< char >& );

template< class T >
class B {
        int data;
        friend void doSomething( B<char>& );
   // declaring function doSomething as friend of B
```

 © Virtual University of Pakistan

```
        …
};

void doSomething( B< char >& cb ) {
        B< int > ib;
        ib.data = 5;    // OK: Accessing private data of class B
        cb.data = 6;    // OK: Accessing private data of class B
}
```

In combined way both can be written as follows,

```
void doSomething( B< char >& );

class A { … };

template< class T > class B {
        int data;
        friend void doSomething( B<char>& );
        friend A;
        …
};
void doSomething( B< char >& cb ) {
        B< int > ib;
        ib.data = 5;    // OK
        cb.data = 6;    // OK
}
class A {
        void method() {
                B< int > ib;
                B< char > cb
                ib.data = 5;    // OK
                cb.data = 6;    // OK
        }
};
```

The Rule 1 is simple. Now, we go to Rule 2.

38.3.Templates and Friends – Rule 2

Rule 2 applies when we make template function or template class friend of another template class.

According to Rule 2 when a friend function template or friend class template is instantiated with the type parameters of class template granting friendship then its instantiation for a specific type is a friend of that class template instantiation for that particular type only. The example below explains this concept,

Consider the code below it has a **template friend function** and a **template friend class** of **template class B**, note that in template class B we have given the **same parameter T** to both template friend function doSomething and template friend class A, according to Rule 2 when we write implementations of friend **template class A**

and friend **template function doSomething** we create instance of class B for same date type for which function **doSomething or class A** will be instantiated as shown below otherwise compiler will generate error,

```
/************************************************************
One template function and one template class
************************************************************/
template< class U >
void doSomething( U ); // template function with template parameter U
template< class V >
class A { … };  // template function with template parameter V

/************************************************************
Making template function and template class friends of class B
Note that in code of class B below we are using same type parameter T for
instantiating template function doSomething and template class A.
************************************************************/
template< class T >
class B {
        int data;
        friend void doSomething( T );   // granting friendship to template
doSomething in // class B
        friend A< T >;  // granting friendship to class A in class B
};

/************************************************************
Implementation of template function doSometing and template class A
As we have used same type parameter T in class B for declaring doSomething and
class A as friend so now we can only instantiate object of class B according to the
passed parameter
************************************************************/

template< class U >
void doSomething( U u ) {    // here template T will be passed as U
        B< U > ib;
   // it is OK as we are instantiating class B for same type that is passed from class B
i.e // T
        ib.data = 78;
}
int main() {
        int i = 5;
        char c = 'x';
        doSomething( i );       // OK
        doSomething( c );       // OK
        return 0;
}
```


© Virtual University of Pakistan

Now, suppose we change instantiation of doSomething and in it we cerate B class instance for int,

```
template< class U >
void doSomething( U u ) {
        B< int > ib;
        ib.data = 78;
}
```

Now, we can only create doSomething instance for integers as shown below, in case we create that function instance for any other date type error will be generated by compiler ,

```
int main() {
        int i = 5;
        char c = 'x';
        doSomething( i );      // OK
        doSomething( c );      // Error!
        return 0;
}
```

The reason for the error is that we are creating class B instance for int data type in function **doSomething(),** but function **doSomething** itself has been instantiated for char in main.

Same Rule applied to template friend class A is shown below,

```
class B< int > {
        int data;
        friend void doSomething( int );
        friend A< int >;
};
template< class T >
class A {
        void method() {
                B< char > cb;    // Error!
                cb.data = 8;
                B< int > ib;  // OK
                ib.data = 9;
        }
};
```

38.4.Templates and Friends – Rule 3

When a friend function / class template takes different 'type parameters' from the class template granting friendship, then its each instantiation is a friend of each

instantiation of the class template granting friendship. Basically here we are removing restriction imposed by Rule 2, due to the use of same type parameter in class B while declaring function doSomething and class A as its friends as shown below,

```
template< class U >
void doSomething( U );
template< class V >
class A { … };


template< class T >
 class B {
        int data;
        template< class W >
        friend void doSomething( W ); // type name is W
        template< class S >
        friend class A; // type name is S
};


template< class U >
void doSomething( U u ) {
        B< int > ib; // Now it is ok to use B for int in function doSomething instantiated for // char in main
        ib.data = 78;
}
```

```
int main() {
        int i = 5;
        char c = 'x';
        doSomething( i );      // OK
        doSomething( c );      // OK
        return 0;
}
```

For classes same Rule is applicable as follows,

```
……..
/*****************************************************************
class A implementation
*****************************************************************/
template< class T >
class A {
        void method() {
                B< char > cb; // OK!
                cb.data = 8;
                B< int > ib;
                ib.data = 9;
        }
```

 © Virtual University of Pakistan

```
};
```

38.5.Templates and Friends – Rule 4

Rule 4 says that when we declare a template as friend of any class then all kinds specializations of that function – explicit, implicit and partial, also becomes friends of the class granting friendship.

```
template< class T >
class B {
        T data;
        template< class U >
                friend class A; // granting friendship to class A for all data types (Rule 3)
};

template< class U >
class A {       // general template class A
        A() {
                B< int > ib;
                ib.data = 10;    // OK
        }
};

template< class U >
class A< U* > {  // template class A specialized for pointer data types
        A() {
                B< int > ib;
                ib.data = 10;    // OK
        }
};
```

In case of functions demonstration of this Rule is below,

```
template< class T >
class B {
        T data;
        template< class U >
                friend void doSomething( U ); // granting friendship to doSomething for all data types (According to Rule 3)
};

template< class U >
void doSomething( U u ) {
        B< int > ib;
        ib.data = 56;    // OK
}

template< >
void doSomething< char >( char u ) {
```

```
        B< int > ib;
        ib.data = 56;    // OK
}
```

Note:

You can see that there is need to concentrate on Rule 2 as it is somewhat difficult to understand other rules are simple.

© Virtual University of Pakistan

Lecture No.39

39.1.Templates & Static Members

We have studied in case of general classes that their static members are created and initialized at class scope, meaning that memory is reserved for them when a class is compiled and they are initialized as well at that time, they are not dependent on the creation of any object of a class, all these tings are true in case of static members of template classes as well with different that they are created when a template class is instantiated for each date type, each instantiation of template class has its own copy of static data members, suppose we have template class with static members, compiler will create this template class implementation for different date types as required, and each implementation will have its own copy of static data members.[16]

Consider the class A with a public static data member and a static member function. (It is not good practice to make any data member public but it is declared as public here to demonstrate that separate copy of this data member will be created for each data type).

```
#include <cstdlib>
#include <iostream>

using namespace std;

template< class T >
class A {
public:
        static int data;
        static void doSomething( T & );
};
template<class T>
int A<T>::data = 0;

int main() {
        A< int > ia;
        A< char > ca;
        ia.data = 5;
        ca.data = 7;
        cout << "ia.data = " << ia.data  << endl  <<  "ca.data = " << ca.data;
        system("PAUSE");
        return 0;
}
```

Long question

Find the output..

Output

ia.data = 5
ca.data = 7

[16] We can take each implementation of template class for a particular data type as a general class written explicitly for that data type.

This output clearly shows that we have two separate copies of data for int and char types.

Also note that each date type instantiation will have single copy of static members as demonstrated below, here objects **ia,ib and ic** will have single copy of **data** being shared by all of them as was the case for static members of a general class,

```
#include <cstdlib>
#include <iostream>

using namespace std;
template< class T >
class A {
public:
        static int data;
        static void doSomething( T & );
};

template<class T>
int A<T>::data = 0;

int main() {
        A< int > ia, ib , ic;

   ia.data = 5;
        ib.data = 7;
   ic.data = 9;
        cout << "ia.data = " << ia.data  << endl  <<  "ib.data = " << ib.data << endl <<
"ic.data = " << ic.data << endl;
        system("PAUSE");
        return 0;
}
```

Output

ia.data = 9
ib.data = 9
ic.data = 9

As same data member is being shared so same member value is being changed

39.2.Templates – Conclusion
Advantages:
Templates provide

- Reusability
- Writability

 © Virtual University of Pakistan

Disadvantages:

- Can consume memory if used without care.
- Templates may affect reliability of a program as they give same type of implementation for all data types with may not be correct for a particular data type like isEqual function given below may give incorrect result for **char ***.

```
template< typename T >
bool isEqual( T x, T y ) {
        return ( x == y );
}
```

- As we are free to give any kind of implementation for a particular data type we may given incorrect implementation, considering the above **isEqual** function, it will produce incorrect result for char * (char arrays) as it will compare first value of both arrays only,

```
int main() {
        char* str1 = "Hello ";
        char* str2 = "World!";
        isEqual( str1, str2 );
                // Compiler accepts!
}
```

39.3.Generic Algorithms Revisited

We studied the concept of Generic Algorithms before that made our code type independent as well as independent of underlying data structure,
For this we developed step by step the following find algorithm that is so generic that it works for all containers,

```
template< typename P, typename T >
P find( P start, P beyond, const T& x ) {
        while ( start != beyond && *start != x )
                ++start;
        return start;
}
```

```
int main() {
        int iArray[5];
        iArray[0] = 15;
        iArray[1] = 7;
        iArray[2] = 987;
        …
        int* found;
        found = find(iArray, iArray + 5, 7);
        return 0;
}
```

39.4.Generic Algorithms Revisited

We claimed that this algorithm is generic, because it works for any aggregate object (container) that defines following three operations

a. Increment operator (++)
b. Dereferencing operator (*)
c. Inequality operator (!=)

Let us now try to apply this generic algorithms to our template **Vector** class to examine whether our generic algorithm **find** works on this container (our template Vector class can store in it data of all types), for this our Vector class need to support the three operations given above and we will have to add one more integer data member index to track the traversal when we will apply increment operation (++), we will also add its setter and getter members to access its value.
So now our modified Vector class is as follows,

Example – Vector

```
template< class T >
class Vector {
private:
        T* ptr;
        int size;
        int index;      // initialized with zero
public:
        Vector( int = 10 );
    Vector( const Vector< T >& );
        T& operator [](int);
        int getIndex() const;
        void setIndex( int i );
        T& operator *();
        bool operator !=( const Vector< T >& v );
        Vector< T >& operator ++();
};
template< class T >
int Vector< T >::getIndex() const {
        return index;
}
template< class T >
void Vector< T >::setIndex( int i ) {
        if ( index >= 0 && index < size )
                index = i;
}
template< class T >
Vector<T>& Vector<T>::operator ++() {
        if ( index < size )
                ++index;
        return *this;
}
template< class T >
T& Vector< T >::operator *() {
```

 © Virtual University of Pakistan

```
        return ptr[index];
}
template< class T >
bool Vector<T>::operator !=( Vector<T>& v ) {
        if ( size != v.size || index != v.index )
                return true;

        for ( int i = 0; i < size; i++ )
                if ( ptr[i] != v.ptr[i] )
                        return true;

        return false;
}
```

Now we want to apply our generic find method on this Vector container, as our generic **find** algorithm is taking three parameters we will create three instances of our Vector class as follows,

1. First Vector instance to store our data (integers in this case).
2. Second Vector instance to pass address of beyond (One position next to first Vector instance)
3. Third Vector instance to get result

```
int main() {
        Vector<int> iv( 3 );
   // First Vector instance to store integer values from which we need to find
integer
   // value
        iv[0] = 10;
        iv[1] = 20;
        iv[2] = 30;

  Vector<int> beyond( iv ),found( 3 );
   // declaring Second (beyond) and third (found) instance in same
statement
   // beyond is initialized with first Vector instance iv
   // third vector instance of length 3 will be used to store the result
        beyond.setIndex( iv.getSize() ); // We have set index of beyond to 3
where no
   // element exists in it so it is behaving as pointing to one element next to
iv.
        found = find( iv, beyond, 20 );
        cout<<"Index: "<<found.getIndex();
        return 0;
}
```

39.5. Generic Algorithm

Now, we want to apply our generic algorithm **find** on this **Vector container**,

```
template< typename P, typename T >
P find( P start, P beyond, const T& x ) {
        while ( start != beyond && *start != x )
                ++start;

        return start;
}
```

This algorithm will run as follows,
First iteration:

Iteration No #	iv size	beyond size	iv index	beyond index	start != beyond	*start != x
1	3	3	0	3	true	true
2	3	3	1	3	true	true
3	3	3	2	3	true	true
4	3	3	3	3	false	undefined

In above scenario this algorithm will run till second iteration only as element 20 will be found iv will be returned with index equal to 1 pointing to second element (20).
As you can see our generic algorithm is working fine with our container **Vector class instance.**
However there are some other things to consider with this generic algorithm implementation.

39.6. Problems

a. Our algorithm doesn't support multiple traversals

We can move forward in single steps that may be very inefficient approach in case of huge Vector object with large number of values, there is no facility to make multiple movements like one in forward direction and at the same time once in reverse direction,

b. Our algorithm shows inconsistent behavior

We generally use pointers to mark a position in a data structure of some primitive type and then return it but here we are using whole container as marker and return it e.g. we returned whole **found** object of Vector class to tell whether we find the value or not we could have done that using single pointer value.

c. Supports only a single traversal strategy

Also there is no way to change traversal strategy (moving beyond more than one value in single step).

© Virtual University of Pakistan

Lecture No.40

Recap

In previous lecture we studied that we can apply Generic Algorithms on any container that supports three operations (++, *, !=)
Then we applied Generic Algorithms **find** on our Vector class by giving implementation of these three operations in **Vector** class
At the end we saw that this approach has some drawbacks as given below,

- No support for multiple traversals
- Supports only a single traversal strategy
- Inconsistent behavior

We saw that we have to made distinct three Vector objects for a simple find method this approach may not be efficient when we have a lot of elements in Vector object.
Now, we see how we can use pointers to modify this approach, we will use three cursors (pointers) to vector class to avoid creating three Vector objects, in this approach we will use pointers to store vector elements addresses (basic concept is similar to link lists).

40.1.Cursors

A better way is to use *cursors.*
A cursor is a pointer that is declared outside the container / aggregate object
Aggregate object provides methods that help a cursor to traverse the elements

imp for mcq , short

- T* first()
- T* beyond()
- T* next(T*)

In this approach, we will add three methods given above in our Vector class that will return pointer to elements of Vector elements instead of whole vector object.
Now, our vector class is as follows,

Vector

```
template< class T >
class Vector {
private:
        T* ptr;
        int size;
public:
        Vector( int = 10 );
        Vector( const Vector< T >& );
        ~Vector();
        int getSize() const;
        const Vector< T >& operator =( const Vector< T >& );
        T& operator []( int );
        T* first();
        T* beyond();
        T* next( T* );
};
template< class T >
T* Vector< T >::first() {
```

```
        return ptr;
}

template< class T >
T* Vector< T >::beyond() {
        return ( ptr + size );
}

template< class T >
T* Vector< T >::next( T* current )
{
        if ( current < (ptr + size) )
                return ( current + 1 );
        // else
        return current;
}

int main() {
        Vector< int > iv( 3 );
        iv[0] = 10;
        iv[1] = 20;
        iv[2] = 30;
        int* first = iv.first();
        int* beyond = iv.beyond();
        int* found = find(first,beyond,20);
        return 0;
}
```

Note our generic find method remained as it as shown below, simply we used cursors (pointers) in our main method to make our code so efficient.

```
template< typename P, typename T >
P find( P start, P beyond, const T& x ) {
        while ( start != beyond && *start != x )
                ++start;
        return start;
}
```

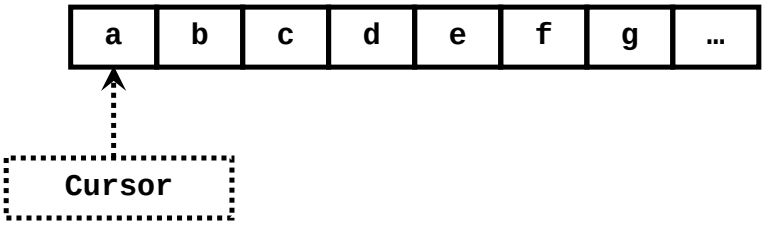
Cursors-Usage

This technique works fine for a contiguous sequence like c++ arrays where elements are placed at consecutive memory locations or Vector whice we have implemented in terms of arrays with extra functionality to overcome shortcomings of c++ arrays, however cursors does now work with containers that use complicated data structures and are non contiguous because in this case we can not simply use ++ operation that we used on our find method, in that case we have to rely on the that container own traversal operations that results in inefficient approach of last lecture,

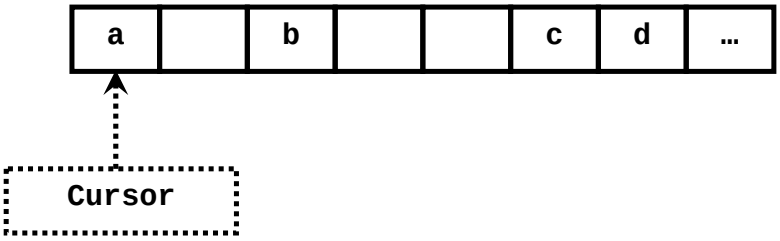
 © Virtual University of Pakistan

Example – Works Fine

This cursor works fine when we have



Example – Problem



Example – Problem

In case of non contiguous container we have issues because when we apply find method on non contiguous containers our find method increment operation fails as elements are not placed at next locations in order, in code below we are calling find method for non contiguous container Set that we are going to study in next lectures in this case our find method will give error as indicated,

```
int main() {
        Set< int > is( 3 );
        is.add( 10 );
        is.add( 20 );
        is.add( 30 );
        ET* first = iv.first();
        ET* beyond = iv.beyond();
        ET* found = find(first, beyond, 20);
        return 0;
}
```

```
template< typename P, typename T >
P find( P start, P beyond, const T& x ) {
        while ( start != beyond && *start != x )
                ++start;        // Error
```

```
        return start;
}
```

In this case we need to use that container own increment operation like shown below, then our find algorithm will work fine,

```
template< typename CT, typename ET >
P find( CT& cont, const ET& x ) {
        ET* start = cont.first();
        ET* beyond = cont.beyond();
        while ( start != beyond && *start != x )
                start = cont.next( start );
        return start;
}
```

```
int main() {
        Set< int > is( 3 );
        is.add( 10 );
        is.add( 20 );
        is.add( 30 );
        int* found = find( is, 20 );
        return 0;
}
```

Cursors – Conclusion

The main benefit is that we are using external pointer so we can do now any kind of traversal in any way i.e. as many traversals as we need only we will need one pointer for each traversal, however we can not use cursors in place of pointers for all containers.

40.2.Iterators

We studied cursors previously, cursors were external pointer that we accessing internal data of any container like vector, it is against the principle of data hiding as we can access any container data using cursors so it is not good programming practice to given access in container for the use of cursors (first, next, beyond methods) we have alternate to cursors in the form of Iterators which are that traverse a container without exposing its internal representation. Mean they are not external pointers but internal data member of container. Iterators are for containers exactly like pointers are for ordinary data structures (you can see this line as we have made a mechanism to declare pointers to our containers like we declare pointers to ordinary data types, in case of cursors we were using ordinary data type pointer but in case of Iterators we will use container pointers without exposing their internal details)

Generic Iterators

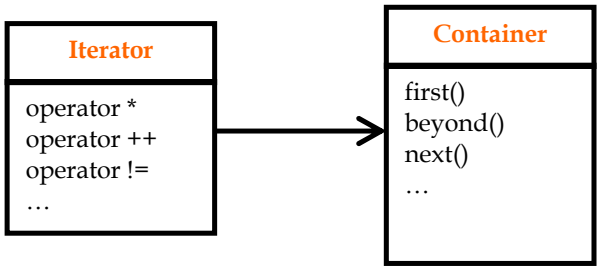
General Iterator class can point to any container because it is implemented using templates, basically it provides us the functionality to create any container object

 © Virtual University of Pakistan

using templates and operator overloading this has been explained in code below, now we will not directly use methods first , beyond and next of container but we will use them through Iterator class i.e we will create Iterator class objects and will iterate container through these methods, however these objects will not be able to directly access internal elements of container. A generic Iterator works with any kind of container. We need the same set of operations in container class to use Iterators,

- T* first()
- T* beyond()
- T* next(T*)

Example – Generic Iterator



Generic Iterator

```
template< class CT, class ET >
// Template Iterator class taking two Vector type parameters one for CT (pointer to
// container type) and ET (pointer to individual element of the container)

class Iterator {
        CT* container;
        ET* index;
public:
        Iterator( CT* c, bool pointAtFirst = true );
        Iterator( Iterator< CT, ET >& it );
        Iterator& operator ++();
        ET& operator *();
        bool operator !=( Iterator< CT, ET >& it );
};

template< class CT, class ET >
Iterator< CT, ET >::Iterator( CT* c, bool pointAtFirst ) {  // constructor
        container = c;
        if ( pointAtFirst )
                index = container->first();
        else
                index = container->beyond();
```

```
}

template< class CT, class ET >
Iterator< CT, ET >::Iterator(        Iterator< CT, ET >& it ) {  // standard copy
constructor
        container = it.container;
        index = it.index;
}

template< class CT, class ET >
Iterator<CT,ET>& Iterator<CT,ET>:: operator ++() {  // increment operator
        index = container->next( index );
        return *this;
}

template< class CT, class ET >
ET& Iterator< CT, ET >::operator *()  // deference operator
{
        return *index;
}

template< class CT, class ET >
bool Iterator< CT, ET >::operator !=( Iterator< CT, ET >& it ) {
// inequality operator result will be false if containers are pointing
// to same memory location or at same index otherwise true
        if ( container != it.container || index != it.index )
                return true;
// else
        return false;
}
```

```
int main() {
        Vector< int > iv( 2 );
        Iterator < Vector<int>, int > it( &iv ) ,  beyond( &iv, false );
   // creating two Iterator objects of type <Vector <int >, int >
   // i.e Iterator will have Vector class object for integers and element type
will be
   // integers so index will be integer pointer
   // first object it is taking Vector class object iv as reference and making
iterate
   //index to point to its first element as bool pointAtFirst is true by default
   // second object beyond is also taking same Vector object as reference
   // so that it may also point to it by as poinAtFrist is false now so index will
point
   // to one position beyond the last element in iv
```


© Virtual University of Pakistan

```
    iv[0] = 10;
        iv[1] = 20;

   Iterator< Vector<int>, int > found        = find( it, beyond, 20 );

    // creating Iterator object found of type <Vector <int >, int > and saving in
it
    // result of found note that now we have saved index of our vector
    // class element if element is found, without creating extra element of
Vector class.

        return 0;
}
```

Note that in above code we are NOT creating multiple objects of Vector class instead we are creating objects of Iterator class against one Vector class object

```
template< typename P, typename T >
P find( P start, P beyond, const T& x ) {
        while ( start != beyond && *start != x )
                ++start;
        return start;
}
```

Iterators – Advantages

a. With Iterators more than one traversal can be pending on a single container
b. Iterators allow to change the traversal strategy without changing the aggregate object
c. They contribute towards data abstraction by emulating pointers

Lecture No.41

41.1.Standard Template Library:

When we started discussing about templates and generic algorithms, our basic aim was to make standard solutions for different problems (like searching or comparison) that should work for all data types and for all containers.
C++ programmers started working on these concepts from very beginning and gave many standard solutions to these problems these standard solutions after the approval of C++ standardization committee were added to a combined library name as Standard Template Library (STL). STL is designed to operate efficiently across many different applications; we can use solutions from this template library in our programs using different header files.

Standard Template Library

STL consists of three key components

very imp mcq

- Containers
- Iterators
- Algorithms

STL Promotes Reuse
STL promotes reuse as we don't need to rewrite the already written standard code for different problems, it saves our development time and cost. Secondly these solutions have been thoroughly tested so there is no change of error due to their use.

41.2.STL Containers

Container is an object that contains a collection of data elements like we have studied before now we will study them in detail.
STL provides three kinds of containers,

very imp mcq

1. Sequence Containers
2. Associative Containers
3. Container Adapters

Sequence Containers

A sequence organizes a finite set of objects, all of the same type, into a strictly linear arrangement

Sequence Containers

a. **vector**

- Rapid insertions and deletions at back end
- Random access to elements

b. **deque**[17]

[17] deque behaves like queue (line) such that we can add elements on both side of it.

 © Virtual University of Pakistan

- Rapid insertions and deletions at front or back
- Random access to elements

c. list[18]

- Doubly linked list
- Rapid insertions and deletions anywhere

Example – STL Vector

```
#include <vector>
int main() {
        std::vector< int > iv;
        int x, y;
        char ch;
        do {
                cout<<"Enter the first integer:";
                cin >> x;
                cout<<"Enter the second
                integer:";
                cin >> y;
                iv.push_back( x );
                iv.push_back( y );
                cout << "Current capacity of iv = " << iv.capacity() << endl;
                cout << "Current size of iv =" << iv.size() << endl;
                cout<<"Do you want to continue?";
                cin >> ch;
        } while ( ch == 'y' );
        return 0;
}
```

Sample Output

```
Enter the first integer: 1

Enter the second integer: 2

Current capacity of iv = 2

Current size of iv = 2

Do you want to continue? y
```

[18] list is linear data structure but can not directly move to any element using index we have to move to a certain by moving sequentially from start element.

Sample Output

```
Enter the first integer: 3

Enter the second integer: 4

Current capacity of iv = 4

Current size of iv = 4

Do you want to continue? y
```

Sample Output

```
Enter the first integer: 5

Enter the second integer: 6

Current capacity of iv = 8

Current size of iv = 6

Do you want to continue? n
```

Example – STL Deque

```
#include <deque>

int main() {
        std::deque< int > dq;
        dq.push_front( 3 );
        dq.push_back( 5 );
        dq.pop_front();
        dq.pop_back()
        return 0;
}
```

Example – STL List

```
#include <list>
int main() {
        std::list< float > _list;
        _list.push_back( 7.8 );
        _list.push_back( 8.9 );
        std::list< float >::iterator it = _list.begin();
```

 © Virtual University of Pakistan

```
        _list.insert( ++it, 5.3 );
        return 0;
}
```

Associative Containers

An associative container provide fast retrieval of data based on keys mean we add elements in these containers using some formula and retriever the elements using the same formula again. It ensures that we do not have to traverse the container one by one but we directly move to the required element for example formula may be,

Value % 10 (remainder function),

Using this formula,

6 will be stored at 6th place
11 will be stored at 1st place
13 will be strored at 3rd place

and so on…

Note that this remainder function will always result in a value between 0 and 9 so if we have an array of size 10 we can use this remainder function easily.
When we need to retrieve a value we will use inverse process to find index of value and will then retrieve it.

Associative Containers
set
No duplicates
multiset
Duplicates allowed
map
No duplicate keys
multimap
Duplicate keys allowed

Example – STL Set

```
#include <set>
int main() {
        std::set< char > cs;
        cout << "Size before insertions: " << cs.size() << endl;
        cs.insert( 'a' );
        cs.insert( 'b' );
        cs.insert( 'b' );
        cout << "Size after      insertions: "
                << cs.size();
        return 0;
```

```
}
```

Output

Size before insertions: 0

Size after insertions: 2

Example – STL Multi-Set

```
#include <set>
int main() {
        std::multiset< char > cms;
        cout << "Size before insertions: " << cms.size() << endl;
        cms.insert( 'a' );
        cms.insert( 'b' );
        cms.insert( 'b' );
        cout << "Size after      insertions: "
                << cms.size();
        return 0;
}
```

Output

Size before insertions: 0

Size after insertions: 3

Example – STL Map

```
#include <map>
int main() {
        typedef std::map< int, char > MyMap;
        MyMap m;
        m.insert(MyMap::value_type(1, 'a'));
        m.insert(MyMap::value_type(2, 'b'));
        m.insert(MyMap::value_type(3, 'c'));
        MyMap::iterator it = m.find( 2 );
        cout << "Value @ key " << it->first << " is " << it->second;
        return 0;
}
```

Output
Value @ key 2 is b

 © Virtual University of Pakistan

Example – STL Multi-Map

```
#include <map>

int main() {
        typedef std::multimap< int, char > MyMap;
        MyMap m;
        m.insert(MyMap::value_type(1, 'a'));
        m.insert(MyMap::value_type(2, 'b'));
        m.insert(MyMap::value_type(3, 'b'));
        MyMap::iterator it1 = m.find( 2 );
        MyMap::iterator it2 = m.find( 3 );
        cout << "Value @ key " << it1->first << " is " << it1->second << endl;
        cout << "Value @ key " << it2->first << " is " << it2->second << endl;
        return 0;
}
```

Output

```
Value @ key 2 is b

Value @ key 3 is b
```

First-class Containers

Sequence and associative containers are collectively referred to as the first-class containers

Container Adapters

A container adapter is a constrained version of some first-class container

Container Adapters

a. **stack**[19]
   - Last in first out (LIFO)
   - Can adapt **vector**, **deque** or **list**
b. **queue**[20]

- First in first out (FIFO)
- Can adapt **deque** or **list**

c. **priority_queue**[21]

[19] Stack is basically a data structure used to store and retrieve values in speedy way because it grows and shrinks from one end only you can consider it like pile of plates present at somewhere plates can be added or retrieved from front end only. It is very efficient approach however it follows last in first out (LIFO) principle meaning things are retrieved in reverse order from stack.

[20] Queue is also data structure and it is similar to queue of people waiting to submit a form or pay their utility bills queue follows first in first out principle (FIFO).

[21] Priority Queue also follows FIFO principle with difference that we add elements in priority queue according to certain priority its example may be printer queue which has priority option the person send

- Always returns element with highest priority
- Can adapt **vector** or **deque**

41.3.Common Functions for All Containers

Common functions that can be applied to all containers are given below (these are basically general functions we can use them with any class),

1. **Default constructor**
2. **Copy Constructor**
3. **Destructor**
4. **empty()**
   - Returns true if container contains no elements
5. **max_size()**
   - Returns the maximum number of elements
6. **size()**
   - Return current number of elements
7. **operator = ()**
   - Assigns one container instance to another
8. **operator < ()**
   - Returns true if the first container is less than the second container
9. **operator <= ()**
   - Returns true if the first container is less than or equal to the second container
10. **operator > ()**
    - Returns true if the first container is greater than the second container
11. **operator >= ()**
    - Returns true if the first container is greater than or equal to the second container
12. **operator == ()**
    - Returns true if the first container is equal to the second container
13. **operator != ()**
    - Returns true if the first container is not equal to the second container
14. **swap ()**
    - swaps the elements of the two containers

41.4.Functions for First-class Containers

As first class containers are both sequence and associative containers both allow fast retrieval of data values so they have following additional functions,

1. **begin()**
   - Returns an iterator object that refers to the first element of the container

print having higher priority will find its data at higher level of the queue than other people having low priority

 © Virtual University of Pakistan

2. **end()**
   - Returns an iterator object that refers to the next position beyond the last element of the container

3. **rbegin()**
   - Returns an iterator object that refers to the last element of the container

4. **rend()**
   - Returns an iterator object that refers to the position before the first element

5. **erase(iterator)**
   - Removes an element pointed to by the iterator

6. **erase(iterator, iterator)**
   - Removes the range of elements specified by the first and the second iterator parameters
7. **clear()**
   - erases all elements from the container

41.5.Container Requirements

As each container need to perform certain operations on the elements added to it like their copy while creating another instance of container or their comparison while performaing certain operation on them like their sorting , so the elements that we are going to add in containers should provide this kind of basic functionality.
Examples of these functionalities are given below,

- When an element is inserted into a container, a copy of that element is made using,
  - Copy Constructor
  - Assignment Operator

So, the elements need to be added to any container should provide copy and assignment functionality. Builtin C++ data types already provide these types of functionalities and we have studied that compiler also generates copy constructor and overloaded assignment operator for our user defined data types like structures and classes if we have not done so.

- Associative containers and many algorithms compare elements so the elements that are added to associative containers should have this functionality,
  - Operator ==
  - Operator <

C++ doesnot provide functionality of comparison operator (==) or less than operator (<) by itself so we have to provide this functionality by ourself in element class if we want to use it in associative containers.

[STL components Iterators and Algorithms will be discussed in next lecture.]

© Virtual University of Pakistan

Lecture No.42

42.1.Iterators

We have studied about Iterators before; they are used to traverse Containers efficiently without accessing internal details, now we see Iterators provided to us in standard template library, Iterators are for containers like pointers are for ordinary data structures STL Iterators provide pointer operations such as * and ++

42.2.Iterator Categories

We can divide Iterators given to us in STL in the following categories,

a. Input Iterators
b. Output Iterators
c. Forward Iterators
d. Bidirectional Iterators
e. Random-access Iterators

Input Iterators

Using input iterators we can read an element, and we can only move in forward direction one element at a time, these can be used in implementing those algorithms that can be solved in one pass (moving container once in single direction from start to end like find algorithm we studied in last lecture).

Output Iterators

Using output iterators we can read an element, and we can only move in forward direction one element at a time, these can be used in implementing those algorithms that can be solved in one pass (moving container once in single direction from start to end like find algorithm we studied in last lecture).

Forward Iterators

Forward iterators have the capabilities of both input and output Iterators, in addition they can bookmark a position in the container (we can set one position as bookmark while traversing the container this will be more understandable when we will see example below)

Bidirectional Iterators

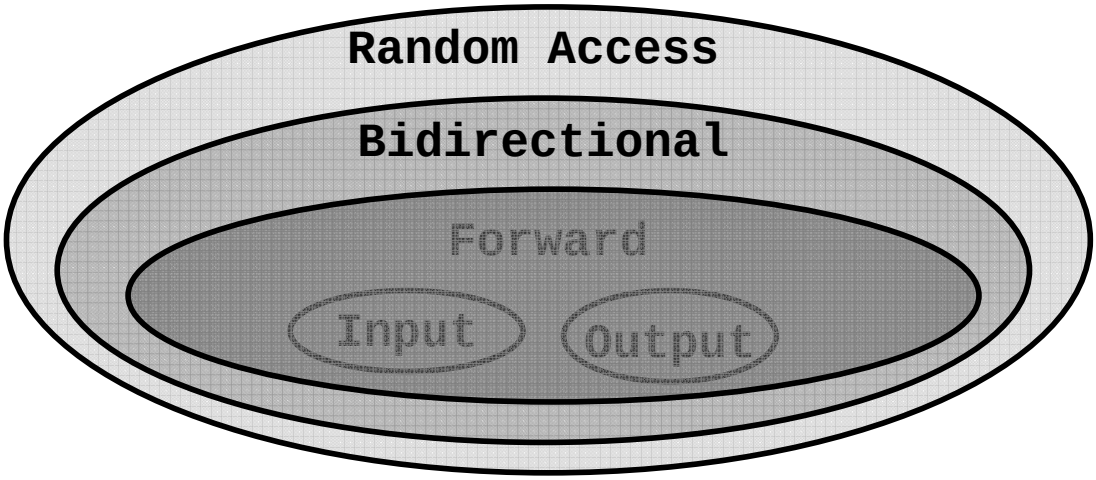
They have all the capabilities of forward Iterators plus they can be moved in backward direction, as a result they support multi-pass algorithms (algorithms that need more that need to traverse container more than once).

Random Access Iterators

They have all the capabilities of bidirectional Iterators plus they can directly access any element of a container.

42.3.Iterator Summary:

Following diagram shows the capabilities and scope of different iterators you can see that Random access iterators have all the capabilities and input and output iterators have least capabilities,



42.4. Container and Iterator Types:

We can use the following different types of iterators with different types of containers, (it is according to the nature of container)

42.5. Sequence Containers

Container Type	Iterator Type	Reason
vector	random access	(as we can access any element of vector using its index so we can use random access iterator)
deque	random access	(in deque we can add elements only in front and back however we can access any element of deque using its index so we can use random access Iterator)
list	bidirectional	(in list we can move in both directions in sequence, however cannot access an element at specific index randomly so we can use bidirectional iterator with list)

 © Virtual University of Pakistan

42.6. Associative Containers

In associative containers we save values based on keys, and we cannot access elements randomly based on indexes as elements are not stored at contiguous memory locations, however, we can traverse them in both directions so, we can use bidirectional iterators with them.

Container Type	Iterator Type	
-- set	--bidirectional	
-- multiset	-- bidirectional	
-- map	-- bidirectional	
-- multimap	-- bidirectional	

42.7. Container Adapters

Container adapters are made with special restrictions, most important restriction is that they don't allow free traversal of their elements so we CANNOT use iterators with them as given below,

Container Type	Iterator Type	
-- stack	-- (none)	
-- queue	-- (none)	
-- priority_queue	-- (none)	

42.8. Iterator Operations

Iterators support following operations,

All Iterators support,

++p
pre-increment an iterator

p++
post-increment an Iterator

Input Iterators support,

***p**
Dereference operator used as rvalue (right value mean they can be used on right side of expression) for reading only, not for assigning value lvalue (left value, assignment only taken place when we are allowed to use them as left value)
p1 = p2
Assignment (two Iterators)
p1 == p2
Equality operator
p1 != p2
Inequality operator
p->

Access Operator

Output Iterators support,

***p**
Dereference operator (here we can use it for assigning new value lvalue as iterator is output iterator that can be used to set any value)

p1 = p2
Assignment

As Forward Iterators have combined properties of both input and output iterators so they support operations of both input and output Iterators.

Bidirectional Iterators

As bidirectional iterators can move in backward direction also, so they support decrementing operations also (moving pointer one element back),

--p
Pre-decrement operator
p--
Post-decrement operator

Random-access Iterators

Besides the operations of bidirectional Iterators, they also support

p + i
Result is an iterator pointing at **p + i**
p – i
Result is an iterator pointing at **p – i**
p += i
Increment iterator p by i positions
p –= i
Decrement iterator p by i positions
p[i]
Returns a reference of element at p + i
p1 < p2
Returns true if **p1** is before **p2** in the container

p1 <= p2
Returns true if **p1** is before **p2** in the container or **p1** is equal to **p2**
p1 > p2
Returns true if **p1** is after **p2** in the container
p1 >= p2
Returns true if **p1** is after **p2** in the container or **p1** is equal to **p2**

 © Virtual University of Pakistan

Example – Random Access Iterator

```
typedef std::vector< int > IntVector;
int main() {
        const int SIZE = 3;
        int iArray[ SIZE ] = { 1, 2, 3 };
        IntVector iv(iArray, iArray + SIZE);
        IntVector::iterator it = iv.begin();
        cout << "Vector contents: ";
        for ( int i = 0; i < SIZE; ++i )
                cout << it[i] << ", ";
        return 0;
}
```

Sample Output

```
Vector contents: 1, 2, 3,
```

Example – Bidirectional Iterator

```
typedef std::set< int > IntSet;
int main() {
        const int SIZE = 3;
        int iArray[ SIZE ] = { 1, 2, 3 };
        IntSet is( iArray, iArray + SIZE        );
        IntSet::iterator it = is.begin();
        cout << "Set contents: ";
        for (int i = 0; i < SIZE; ++i)
                cout << it[i] << ", "; // Error
        return 0;
}
```

Example – Bidirectional Iterator

```
typedef std::set< int > IntSet;
int main() {
        const int SIZE = 3;
        int iArray[ SIZE ] = { 1, 2, 3 };
        IntSet is( iArray, iArray + SIZE        );
        IntSet::iterator it = is.begin();
        cout << "Set contents: ";
        for ( int i = 0; i < SIZE; ++i )
                cout << *it++ << ", ";   // OK
        return 0;
}
```

Sample Output

```
Set contents: 1, 2, 3,
```

Example – Bidirectional Iterator

```
typedef std::set< int > IntSet;
int main() {
        const int SIZE = 3;
        int iArray[ SIZE ] = { 1, 2, 3 };
        IntSet is( iArray, iArray + SIZE );
        IntSet::iterator it = is.end();
        cout << "Set contents: ";
        for (int i = 0; i < SIZE; ++i)
                cout << *--it << ", ";
        return 0;
}
```

Sample Output

```
Set contents: 3, 2, 1,
```

Example – Input Iterator

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <iterator>

int main() {
        int x, y, z;
        cout << "Enter three integers:\n";
        std::istream_iterator< int > inputIt( cin );
        x = *inputIt++;
        y = *inputIt++;
        z = *inputIt;
        cout << "x = " << x << endl;
        cout << "y = " << y << endl;
        cout << "z = " << z << endl;
        return 0;
}
```

```
int main() {
```

 © Virtual University of Pakistan

```
        int x = 5;
        std::istream_iterator< int > inputIt( cin );
        *inputIt = x;    // Error
        return 0;
}
```

Example – Output Iterator

```
int main() {
        int x = 1, y = 2, z = 3;
        std::ostream_iterator< int > outputIt( cout, ", " );
        *outputIt++ = x;
        *outputIt++ = y;
        *outputIt++ = z;
        return 0;
}
```

Example – Output Iterator

```
int main() {
        int x = 1, y = 2, z = 3;
        std::ostream_iterator< int > outputIt( cout, ", " );
        x = *outputIt++;        // Error
        return 0;
}
```

42.9. Algorithms

STL includes 70 standard algorithms

These algorithms may use Iterators to manipulate containers

STL algorithms also work for ordinary pointers and data structures

An algorithm works with a particular container only if that container supports a particular Iterator category

A multi-pass algorithm for example, requires bidirectional Iterator(s) at least

Algorithm: Examples

Mutating-Sequence Algorithms (that required changing of elements position)

Copy

```
        copy_backward
        fill
        fill_n
        generate
        generate_n
        iter_swap
        partition
    ……
```

Non-Mutating-Sequence Algorithms (that don't require changing of element position)

```
        adjacent_find
        count
        count_if
        equal
        find
        find_each
        find_end
        find_first_of
        …
```

Numeric Algorithms (involves mathematical calculation)

accumulate

inner_product

partial_sum

adjacent_difference

Example – **copy** Algorithm

```
#include <iostream>
using std::cout;
#include <vector>
#include <algorithm>
typedef std::vector< int > IntVector;

int main() {
        int iArray[] = {1, 2, 3, 4, 5, 6};
        IntVector iv( iArray, iArray + 6 );
std::ostream_iterator< int >
                                output( cout, ", " );
        std::copy( begin, end, output );

        return 0;
}
```

 © Virtual University of Pakistan

Output

```
1, 2, 3, 4, 5, 6,
```

Example – **fill** Algorithm

```
#include <iostream>
using std::cout;
using std::endl;
#include <vector>
#include <algorithm>
typedef std::vector< int > IntVector;
int main() {
        int iArray[] = { 1, 2, 3, 4, 5 };
        IntVector iv( iArray, iArray + 5 );
std::ostream_iterator< int > output( cout, ", " );
        std::copy( iv.begin(), iv.end(), output );
        std::fill(iv.begin(), iv.end(), 0);
        cout << endl;
        std::copy( iv.begin(), iv.end(), output );
        return 0;
}
```

Lecture No.43

Techniques for Error Handling:

Sometimes our program terminates abnormally, sometimes they even crash the system, these errors occur mostly due to incorrect memory access or due to input/output error, sometimes it is our program fault and sometimes it is some external resource error (like network or hard disk).
If we allow these errors to happen we may lose our work for example if a text editor terminates abnormally without allowing us to save our work we will lost our work, so it is important that we add some type of error handling mechanism in our program, we use the following techniques for error handling,

a. Abnormal termination
b. Graceful termination
c. Return the illegal value
d. Return error code from a function
e. Exception handling

43.1. Example – Abnormal Termination
In abnormal termination we do nothing and our program is terminated abnormally by operating system if case of any error without saving program data,

```
void GetNumbers( int &a, int &b ) {
        cout << "\nEnter two integers";
        cin >> a >> b;
}
int Quotient( int a, int b ){
        return a / b;
}
void OutputQuotient( int a, int b, int quo ) {
        cout << "Quotient of " << a << " and "
                << b << " is " << quo << endl;
}

int main(){
        int sum = 0, quot;
        int a, b;
        for (int i = 0; i < 10; i++){
                GetNumbers(a,b);
                quot = Quotient(a,b);
                sum += quot;
                OutputQuotient(a,b,quot);
        }
        cout << "\nSum of ten quotients is "<< sum;
        return 0;
}
```

 © Virtual University of Pakistan

Output

```
Enter two integers
10
10
Quotient of 10 and 10 is 1
Enter two integers
10
0
Program terminated abnormally
```

43.2. Graceful Termination

Program can be designed in such a way that instead of abnormal termination, that causes the wastage of resources, program performs clean up tasks, mean we add check for expected errors (using if conditions),

Example – Graceful Termination

```
int Quotient (int a, int b ) {
        if(b == 0){
                cout << "Denominator can't " << " be zero" << endl;
                // Do local clean up
                exit(1);
        }
        return a / b;
}
```

Output

```
Enter two integers
10
10
Quotient of 10 and 10 is 1
Enter two integers
10
0
Denominator can't be zero
```

43.3. Error Handling

a. Return Illegal Value

The clean-up tasks are of local nature only; there remains the possibility of information loss.

Example – Return Illegal Value

```
int Quotient(int a, int b){
        if(b == 0)
            b = 1;
        OutputQuotient(a, b, a/b);
        return a / b ;
}
int main() {
        int a,b,quot;    GetNumbers(a,b);
        quot = Quotient(a,b);
        return 0;
}
```

Output

```
Enter two integers
10
0
Quotient of 10 and 1 is 10
```

b. Return Error Code

Programmer has avoided the system crash but the program is now in an inconsistent state

Example – Return Error Code

```
bool Quotient ( int a, int b, int & retVal ) {
        if(b == 0){
            return false;
        }
        retVal = a / b;
        return true;
}
```

Part of main Function

```
for(int i = 0; i < 10; i++){
        GetNumbers(a,b);
        while ( ! Quotient(a, b, quot) ) {
            cout << "Denominator can't be " << "Zero. Give input again \n";
            GetNumbers(a,b);
        }
        sum += quot;
        OutputQuotient(a, b, quot);
```

 © Virtual University of Pakistan

```
}
```

Output

```
Enter two integers
10
0
Denominator can't be zero. Give input again.
Enter two integers
10
10
Quotient of 10 and 10 is 1
...//there will be exactly ten quotients
```

Error Handling

- Issues in Error Handling:
- Programmer sometimes has to change the design to incorporate error handling
- Programmer has to check the return type of the function to know whether an error has occurred
- Programmer of calling function can ignore the return value
- The result of the function might contain illegal value, this may cause a system crash later
- Program's Complexity Increases
  The error handling code increases the complexity of the code
  - Error handling code is mixed with program logic
  - The code becomes less readable
  - Difficult to modify

The example below shows these concepts,

Example- without error handling

```
int main() {
        function1();
        function2();
        function3();
        return 0;
}
```

Example – with error handling

```
int main(){
        if( function1() ) {
```

```
                    if( function2() ) {
                        if( function3() ) {
                            ...
                        }
                        else    cout << "Error Z has occurred";
                    }
                    else    cout << "Error Y has occurred";
                }
                else    cout << "Error X has occurred";
                return 0;
}
```

43.4. Exception Handling

- Exception handling is a much elegant solution as compared to other error handling mechanisms
- It enables separation of main logic and error handling code

43.5. Exception Handling Process

- Programmer writes the code that is suspected to cause an exception in **try block**
- Code section that encounters an error **throws an object** that is used to represent exception
- **Catch blocks** follow try block to catch the object thrown

Syntax – Throwing an exception

- The keyword throw is used to throw an exception
- Any expression can be used to represent the exception that has occurred
    - throw X;
    - throw (X);

Examples

```
int a;
Exception obj;
throw 1;                // throw with literal (const)
throw (a);              // throw with variable
throw obj;              // throw with object
throw Exception(); // anonymous  object
throw 1+2*9; // mathematical expression
```

- Primitive data types may be avoided as throw expression, as they can cause ambiguity
- Define new classes to represent the exceptions that has occurred
    - This way there are less chances of ambiguity

© Virtual University of Pakistan

Syntax – Try and Catch

```
int main () {
        try {
                ... exception with be thrown here in case of error
        }
        catch ( Exception1 ) {    // exception with be caught here if exception was thrown
                ...
        }
        catch ( Exception2 obj ) { // exception with be caught here if exception was
    // thrown
                ...
        }
        return 0;
}
```

Catch Blocks

- Catch handler must be preceded by a try block or an other catch handler
- Catch handlers are only executed when an exception has occurred
- Catch handlers are differentiated on the basis of argument type
- The catch blocks are tried in order they are written
- They can be seen as switch statement that do not need break keyword

Complete Example of try catch and throw:

```
class DivideByZero { // class just use to indicate and throw an exception
public:
        DivideByZero() {
        }
};
int Quotient(int a, int b){
        if(b == 0){
                throw DivideByZero();  // throwing above class object as exception in
          // case of error
        }
        return a / b;
}
```

Related part of main getting user input and calling Quotient method in try catch block this method will throw exception (DividebyZero class object) if user enter no. b (denominator, divider) equal to zero and this exception will be caught in catch block below, (you can write it as complete program to test the actual working of program)

```
for(int i = 0; i < 10; i++) {
        try{
                GetNumbers(a,b);
                quot = Quotient(a,b);
                OutputQuotient(a,b,quot); sum += quot;
        }
        catch(DivideByZero) {
                i--;
                cout << "\nAttempt to divide
    numerator with zero";
        }
}
```

Output

```
Enter two integers
10
10
Quotient of 10 and 10 is 1
Enter two integers
10
0
Attempt to divide numerator (dividend) with zero
...
```

Catch Handler

- The catch handler catches the DivideByZero exception through anonymous object
- Program logic and error handling code are separated
- We can modify this to use the object to carry information about the cause of error

Separation of Program Logic and Error Handling

You can see now that error handling code is separated from man logic of program as shown below,

```
int main() {
        try {
                function1();
                function2();
                function3();
```

 © Virtual University of Pakistan

```
        }
        catch( ErrorX) { ... }
        catch( ErrorY) { ... }
        catch( ErrorZ) { ... }
        return 0;
}
```

Lecture No.44

Previous Lecture Example of Exception Handling:

```
class DivideByZero {
public:
        DivideByZero() {
        }
};
int Quotient(int a, int b){
        if(b == 0){
                throw DivideByZero();
        }
        return a / b;
}

//main Function

int main() {
        try{    …
                quot = Quotient(a,b);
                …
        }
        catch(DivideByZero) {
                …
        }
        return 0;
}
```

Now, we want to see what happens actually to local variables in try block when an exception is thrown, this concept is called stack unwinding which tells how try catch blocks are un winded (executed) when there are nested function calls involving try catch blocks or nested try catch blocks themselves.

44.1. Stack Unwinding

The flow control (the order in which code statements and function calls are made) as a result of throw statement is referred as "stack unwinding"

Stack Unwinding can take place in the following two ways,

1. When we have nested try catch blocks (one try catch block into other try catch block), for example

```
try {
        try {

    } catch( Exception e)  {

  }
```


© Virtual University of Pakistan

```
 } catch(exception e){

}
```

2. When exception is thrown from nested functions having try catch blocks

```
void function1() {

        throw Exception();
}
void function2() {

        function1();

}

int main() {
        try{
                function2();
        } catch( Exception ) { }
        return 0;
}
```

Stack unwinding is more complex than simple nested function calls (or recursive function calls) as in case of nested try catch block, exception can be thrown from any try block, so transfer of control to catch handler is complex, we see this is detail,

First note these points,

- All the local objects of an executing block are destroyed when an exception is thrown
- Dynamically allocated memory is not destroyed automatically
- If no catch handler catches the exception the function terminate is called, which by default calls function abort

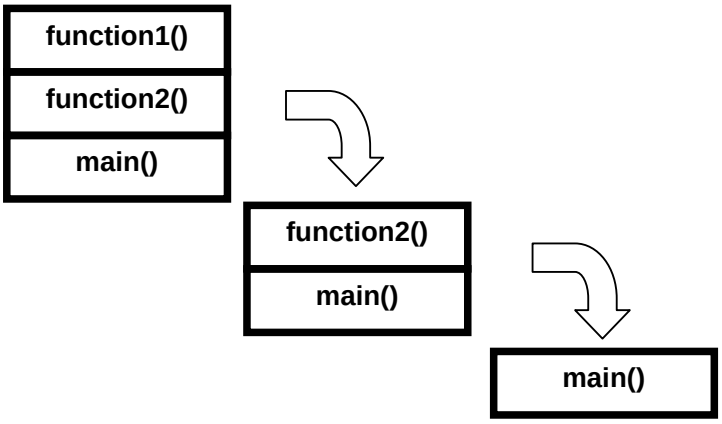
Examples

Nested Functions example:

In example below we have two functions function1 and function2, function2 is calling function1, in function1 we have added exception throwing code so it is necessary now to call function1 in try catch blocks otherwise compiler will generated an error, we are calling function2 in main, now note that function2 ifself is calling function1 that needs try catch block so we need to call function2 in try catch block, (otherwise compiler will generate an error), in case function1 code generates an exception stack unwinding takes place control will be returned to function2 which will return control to main the diagram below code explain this concept,

```
void function1() {
        throw Exception();
}
void function2() {
```

```
                function1();
}
int main() {
                try{
                        function2();
                } catch( Exception ) { }
                return 0;
}
```



Nested Try catch blocks example:

The stack unwinding is also performed if we have nested try catch blocks,

Example

```
int main( ) {
                try {
                        try {
                                throw 1;
                        }
                        catch( float ) { }
                }
                catch( int ) {
                 }
                return 0;
}
```

When an exception will be thrown from a try block control will go to catch block of that try block if it is not appropriate catch block then control will go to ctach blocks of other try blocks above it one by one, Stack Unwinding will occur in following order in above example,

© Virtual University of Pakistan

- **If exception is thrown from innermost try block,**

- Firstly the catch handler with float parameter is tried (innermost), this catch handler will not be executed as its parameter is of different type – no coercion (match)
- Secondly the catch handler with int parameter is tried and executed

- **If exception is thrown from outer try block, then as there is no other try block** above it so only this block catch handler will be matched with exception if it matches catch block will be exectuted otherwise default terminate and abort functions (discussed in this lecture start) will be called.

Catch Handler

- We can modify the code in catch handler to use the exception object to carry information about the cause of error
- The exception object thrown is copied to the object given in the handler
- We pass the exception as reference instead of by value in the catch handler to avoid problem caused by shallow copy

Example

We have added a method Print in our exception class that to show the user cause of error,

```
class DivideByZero { // exception class
        int numerator;
public:
        DivideByZero(int i) {  // constructor taking one parameter (dividend)
                numerator = i;
        }
        void Print() const{
                cout << endl << numerator
                << " was divided by zero";
        }
};
int Quotient(int a, int b) {
        if(b == 0){
                throw DivideByZero(a);
        }
        return a / b;
}
for ( int i = 0; i < 10; i++ ) {
        try  {
                GetNumbers(a, b);
                quot = Quotient(a, b); ...
        } catch(DivideByZero & obj) {
```

```
                    obj.Print();
                    i--;
                }
}
```

Body of main Function
Output

```
Enter two integers
10
10
Quotient of 10 and 10 is 1
Enter two integers
10
0
10 was divided by zero
...
```

Catch Handler

The object thrown as exception is destroyed when the execution of the catch handler completes

Avoiding too many Catch Handlers

There are two ways to catch more then one object in a single catch handler

- Use inheritance
- Catch every exception

Inheritance of Exceptions

In inheritance we Group all exceptions according to their categories and catch single exception for whole category for example for code below we have divided the exceptions as follows,

- Math exceptions (Divide by Zero and IntegerOutOfRange exception)
- Input Output exceptions(InputStreamError)

```
try{
                ...
}
catch(DivideByZero){
                ...
}
catch(IntergerOutOfRange){
                ...
}
catch (InputStreamError){
```


© Virtual University of Pakistan

```
}
```

Example–With Inheritance

```
try{
                ...
}
catch (MathError){
}
catch (InputStreamError){
}
```

Catch Every Exception
C++ provides a special syntax that allows to catch every object thrown

```
catch ( … )
{
                //...
}
```

Re-Throw

A function can catch an exception and perform partial handling
Re-throw is a mechanism of throw the exception again after partial handling

```
        throw; /*without any expression*/
```

Example

```
int main ( ) {
                try {
                        Function();
                }
                catch(Exception&) {
                        ...
                }
                return 0;
}
void Function( ) {
                try {
                        /*Code that might throw
                                an Exception*/
                       }

            catch(Exception&) {
                        if( can_handle_completely ) {
                                // handle exception
                        } else {
```

```
                                    // partially handle exception
                                    throw; //re-throw exception
                                }
                            } // end of catch
} // end of function
```

Order of Handlers

Order of the more than one catch handlers can cause logical errors when using inheritance or catch all (however compiler will not generate any error in this case)

Example

```
try{
                    ...
}
catch (...) {                   ...
}
catch ( MathError ) {  ...
}
catch ( DivideByZero ) {        ...
}
// last two handlers can never be invoked as general exception class will catch all
// exceptions including the next two
```


© Virtual University of Pakistan

Lecture No.45

45.1.Resource Management

- Function acquiring a resource must properly release it
- Throwing an exception can cause resource wastage

Example

```
int function1(){
        FILE *fileptr = fopen("filename.txt","w");
        ...
        throw exception();
        ...
        fclose(fileptr);
        return 0;
}
```

In case of exception the call to fclose will be ignored and file will remain opened.

We can remove this issue in following ways,

First Attempt

```
int function1(){
        try{
                FILE *fileptr = fopen("filename.txt","w");
                fwrite("Hello  World",1,11,fileptr);
                ...
                throw exception();
                fclose(fileptr);
        } catch(...) {
                fclose(fileptr);  // adding fclose in catch handler as well
                throw;
        }
        return 0;
}
```

But it is,code duplication.

Second Attempt:
Adding a separate class file handling what will have constructor to open file and destructor to close it, as you know objects and local variables in try block are destroyed automatically when try block complete its execution or in case exception is thrown, so this file object will automatically be destroyed,

```
class FilePtr{
```

```
                FILE * f;
public:
                FilePtr(const char *name, const char * mode) {
                        f = fopen(name, mode);
                    }
                ~FilePtr() {
                        fclose(f);
                    }
                operator FILE * () {
                         return f;
                     }
};

int function1(){
                FilePtr file("filename.txt","w");
                fwrite("Hello  World",1,11,file);
                throw exception();
                ...
                return 0;
}
```

- The destructor of the FilePtr class will close the file
- Programmer does not have to close the file explicitly in case of error as well as in normal case

Exception in Constructors

Exception thrown in constructor cause the destructor to be called for any object built as part of object being constructed before exception is thrown
Destructor for partially constructed object is not called

Example

```
class Student{
                String FirstName;
                String SecondName;
                String EmailAddress;
                …
};
```

If the constructor of the SecondName throws an exception then the destructor for the First Name will be called.

So, generally we can say that in constructor if an exception is thrown than all objects created so far are destroyed, if EmailAddress String object had thrown exception then SecondName and FirstName objects will be destroyed using their destructor. However destructor of Student class itself will not be called in any case as its object was not completely constructed.

 © Virtual University of Pakistan

Exception in Initialization List

Exception due to constructor of any contained object or the constructor of a parent class can be caught in the member initialization list

Example

```
Student::Student (String aName) : name(aName)
/*The constructor of String can throw a exception*/
{
        ...
}
```

Exception in Initialization List

The programmer may want to catch the exception and perform some action to rectify the problem

Example

```
Student::Student (String aName)
    try     : name(aName) {
            ...
    }
    catch(…) {
    }
```

Exceptions in Destructors

Exception should not leave the destructor, because when a destructor is running it means that we have a stack unwinding going on that has run this destructor to delete this object if this exception will be allowed to run it will run another stack unwinding mechanism and this will be called leaving the exception from destructor and is not allowed. C++ allows running only one stack unwinding process at a time.
If a destructor is called due to stack unwinding, and an exception leaves the destructor then the function std::terminate() is called, which by default calls the std::abort()

Example

```
class Exception;
class Complex{
        …
public:
        ~Complex(){
                throw Exception();
        }
};
int main(){
        try{
```

```
                Complex obj;
                throw Exception();
                …
        }
        catch(…){
        }
        return 0;
}

// The program will terminate abnormally
```

So add catch exception in destructor itself as shown below, in this case single stack unwinding process may handle the situation.

Example

```
Complex::~Complex()
{
        try{
                throw Exception();
        }
        catch(…){
        }
}
```

Exception Specification

- Program can specify the list of exceptions a function is allowed to throw
- This list is also called throw list
- If we write empty list then the function wont be able to throw any exception

Syntax

```
void Function1() {…}
void Function2() throw () {…}
void Function3( )   throw (Exception1, …){}
```

Here,

- Function1 can throw any exception
- Function2 cannot throw any Exception
- Function3 can throw any exception of type Exception1 or any class derived from it

Exception Specification

- If a function throws exception other then specified in the throw list then the function unexpected is called

 © Virtual University of Pakistan

- The function unexpected calls the function terminate and terminates the program
- If programmer wants to handle such cases also then he must provide a handler function and tell the compiler to call handler using set_unexpected

Course Review

We have studied the following topics in this course,

Object Orientation

- What is an object
- Object-Oriented Model
  - Information Hiding
  - Encapsulation
  - Abstraction
- Classes

Object Orientation

- Inheritance
  - Generalization
  - Sub-Typing
  - Specialization
- "IS-A" relationship
- Abstract classes
- Concrete classes

Object Orientation

- Multiple inheritance
- Types of association
  - Simple association
  - Composition
  - Aggregation
- Polymorphism

Classes – C++ Constructs

- Classes
  - Data members
  - Member functions
- Access specifier
- Constructors
- Copy Constructors
- Destructors

Classes – C++ Constructs

- this pointer

- Constant objects
- Static data member
- Static member function
- Dynamic allocation

Classes – C++ Constructs

- Friend classes
- Friend functions
- Operator overloading
    - Binary operator
    - Unary operator
    - operator []
    - Type conversion

Inheritance – C++ Constructs

- Public inheritance
- Private inheritance
- Protected inheritance
- Overriding
- Class hierarchy

Polymorphism – C++ Constructs

- Static type vs. dynamic type
- Virtual function
- Virtual destructor
- V-tables
- Multiple inheritance
- Virtual inheritance

Templates – C++ Constructs

- Generic programming
- Classes template
- Function templates
- Generic algorithm
- Templates specialization
    - Partial Specialization
    - Complete specialization

Templates – C++ Constructs

- Inheritance and templates
- Friends and templates
- STL
    - Containers
    - Iterators

 © Virtual University of Pakistan

- Algorithms

Writing Reliable Programs

- Error handling techniques
  - Abnormal termination
  - Graceful termination
  - Return the illegal value
  - Return error code from a function
  - Exception handling

Note:

Please give your suggestions/comments about this document at the email address cs304@vu.edu.pk, in the following format

Page No.#	Issue/Error