Object-Oriented Programming
(OOP)
Lecture No. 1

Course Objective

- Objective of this course is to make students familiar with the concepts of object-oriented programming

- Concepts will be reinforced by their implementation in C++

Course Contents

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

Books

- ►C++ How to Program
  By Deitel & Deitel
- ►The C++ Programming Language
  By Bjarne Stroustrup
- ►Object-Oriented Software Engineering
  By Jacobson, Christerson, Jonsson, Overgaard

Grading Policy

► Assignments	15 %
► Group Discussion	5 %
► Mid-Term	35 %
► Final	45 %

Object-Orientation (OO)

What is Object-Orientation?

- ►A technique for system modeling
- ►OO model consists of several interacting objects

What is a Model?

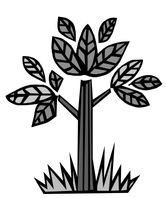
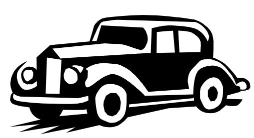
- ►A model is an abstraction of something
- ►Purpose is to understand the product before developing it

Examples – Model

- ►Highway maps
- ►Architectural models
- ►Mechanical models

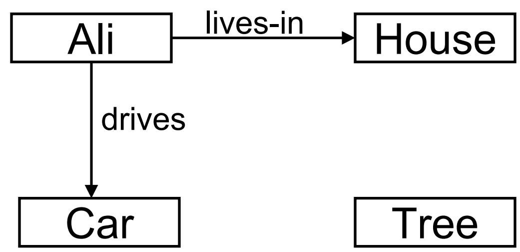
Example – OO Model



...Example – OO Model

- Objects
  - Ali
  - House
  - Car
  - Tree
- Interactions
  - Ali lives in the house
  - Ali drives the car



Object-Orientation - Advantages

- People think in terms of objects
- OO models map to reality
- Therefore, OO models are
  - easy to develop
  - easy to understand

What is an Object?

An object is

► Something tangible (Ali, Car)

► Something that can be apprehended intellectually (Time, Date)

... What is an Object?

An object has

► State (attributes)
► Well-defined behaviour (operations)
► Unique identity

Example – Ali is a Tangible Object

- ► State (attributes)
  - Name
  - Age
- ► behaviour (operations)
  - Walks
  - Eats
- ► Identity
  - His name

Example – Car is a Tangible Object

- ► State (attributes)
  - Color
  - Model
- ► behaviour (operations)
  - Accelerate
  - Start Car
  - Change Gear
- ► Identity
  - Its registration number

Example – Time is an Object Apprehended Intellectually

- State (attributes)
  - Hours
  - Seconds
  - Minutes
- behaviour (operations)
  - Set Hours
  - Set Seconds
  - Set Minutes
- Identity
  - Would have a unique ID in the model

Example – Date is an Object Apprehended Intellectually

- State (attributes)
  - Year
  - Day
  - Month
- behaviour (operations)
  - Set Year
  - Set Day
  - Set Month
- Identity
  - Would have a unique ID in the model

Object-Oriented Programming
(OOP)
Lecture No. 2

Information Hiding

► Information is stored within the object

► It is hidden from the outside world

► It can only be manipulated by the object itself

Example – Information Hiding

- ►Ali’s name is stored within his brain
- ►We can’t access his name directly
- ►Rather we can ask him to tell his name

Example – Information Hiding

- ►A phone stores several phone numbers
- ►We can’t read the numbers directly from the SIM card
- ►Rather phone-set reads this information for us

Information Hiding Advantages

- ►Simplifies the model by hiding implementation details
- ►It is a barrier against change propagation

Encapsulation

- ►Data and behaviour are tightly coupled inside an object
- ►Both the information structure and implementation details of its operations are hidden from the outer world

Example – Encapsulation

- ►Ali stores his personal information and knows how to translate it to the desired language
- ►We don't know
  - How the data is stored
  - How Ali translates this information

Example – Encapsulation

- ►A Phone stores phone numbers in digital format and knows how to convert it into human-readable characters
- ►We don't know
  - How the data is stored
  - How it is converted to human-readable characters

Encapsulation – Advantages

- ►Simplicity and clarity
- ►Low complexity
- ►Better understanding

Object has an Interface

- ►An object encapsulates data and behaviour
- ►So how objects interact with each other?
- ►Each object provides an interface (operations)
- ►Other objects communicate through this interface

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

Implementation

- ►Provides services offered by the object interface
- ►This includes
  - Data structures to hold object state
  - Functionality that provides required services

Example – Implementation of Gear Box

- ►Data Structure
  - Mechanical structure of gear box
- ►Functionality
  - Mechanism to change gear

Example – Implementation of Address Book in a Phone

► Data Structure
  - SIM card

► Functionality
  - Read/write circuitry

Separation of Interface & Implementation

► Means change in implementation does not effect object interface

► This is achieved via principles of information hiding and encapsulation

Example – Separation of Interface & Implementation

- ►A driver can drive a car independent of engine type (petrol, diesel)
- ►Because interface does not change with the implementation

Example – Separation of Interface & Implementation

- ►A driver can apply brakes independent of brakes type (simple, disk)
- ►Again, reason is the same interface

Advantages of Separation

- ►Users need not to worry about a change until the interface is same
- ►Low Complexity
- ►Direct access to information structure of an object can produce errors

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

Object-Oriented Programming
(OOP)
Lecture No. 3

Abstraction

- Abstraction is a way to cope with complexity.
- Principle of abstraction:

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

Example – Abstraction

Ali is a PhD student and teaches BS students

► Attributes

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

Example – Abstraction

Ali is a PhD student and teaches BS students

► behaviour

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

Example – Abstraction

Student’s Perspective

► Attributes

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

Example – Abstraction

Student’s Perspective

► behaviour

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

Example – Abstraction

Teacher’s Perspective

► Attributes

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

Example – Abstraction

Teacher’s Perspective

► behaviour

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

Example – Abstraction

A cat can be viewed with different perspectives

► Ordinary Perspective

A pet animal with
- Four Legs
- A Tail
- Two Ears
- Sharp Teeth

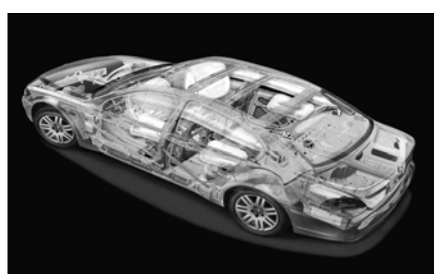
► Surgeon's Perspective

A being with
- A Skeleton
- Heart
- Kidney
- Stomach

Example – Abstraction

Driver's View

Engineer's View



Abstraction – Advantages

- Simplifies the model by hiding irrelevant details
- Abstraction provides the freedom to defer implementation decisions by avoiding commitment to details

Classes

- In an OO model, some of the objects exhibit identical characteristics (information structure and behaviour)
- We say that they belong to the same class

Example – Class

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

- Each one is a Student
- We say these objects are *instances* of the Student class

Example – Class

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

- Each one is a teacher
- We say these objects are *instances* of the Teacher class

Graphical Representation of Classes

(Class Name)
(attributes)
(operations)

Normal Form

(Class Name)

Suppressed Form

Example – Graphical Representation of Classes

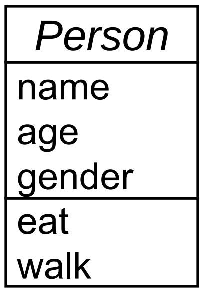
Circle
center radius
draw computeArea

Normal Form

Circle

Suppressed Form

Example – Graphical Representation of Classes



Normal Form

Suppressed Form

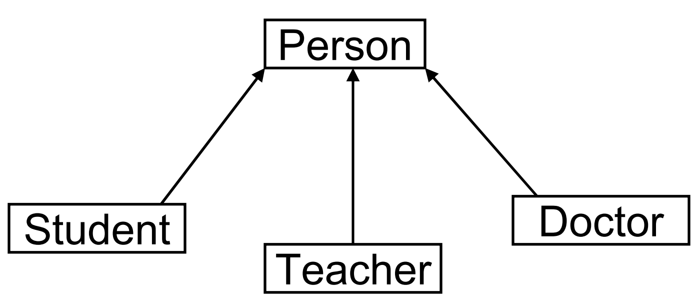
Inheritance

- ►A child inherits characteristics of its parents
- ►Besides inherited characteristics, a child may have its own unique characteristics

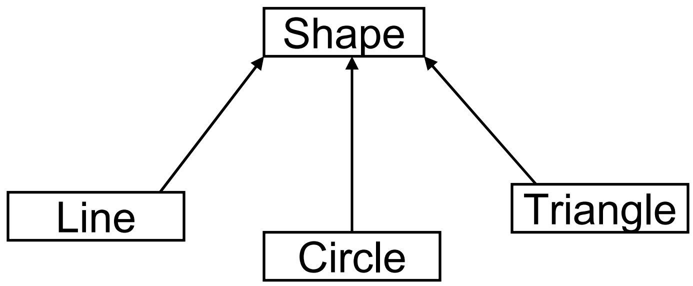
Inheritance in Classes

- 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

Example – Inheritance



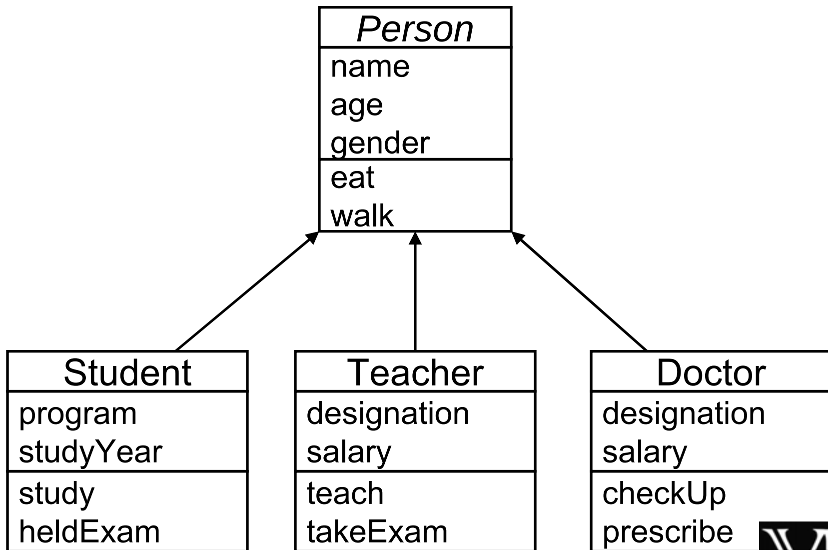
Example – Inheritance



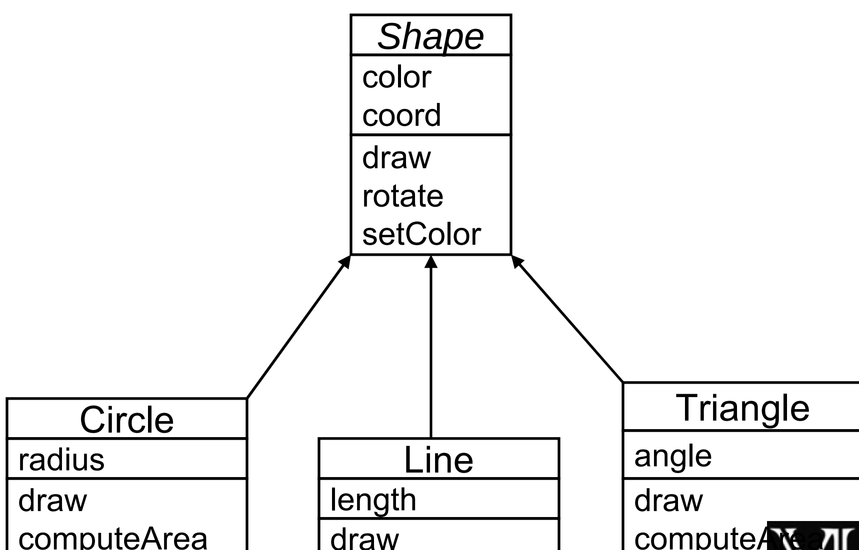
Inheritance – “IS A” or “IS A KIND OF” Relationship

► Each derived class is a special kind of its base class

Example – “IS A” Relationship



Example – “IS A” Relationship



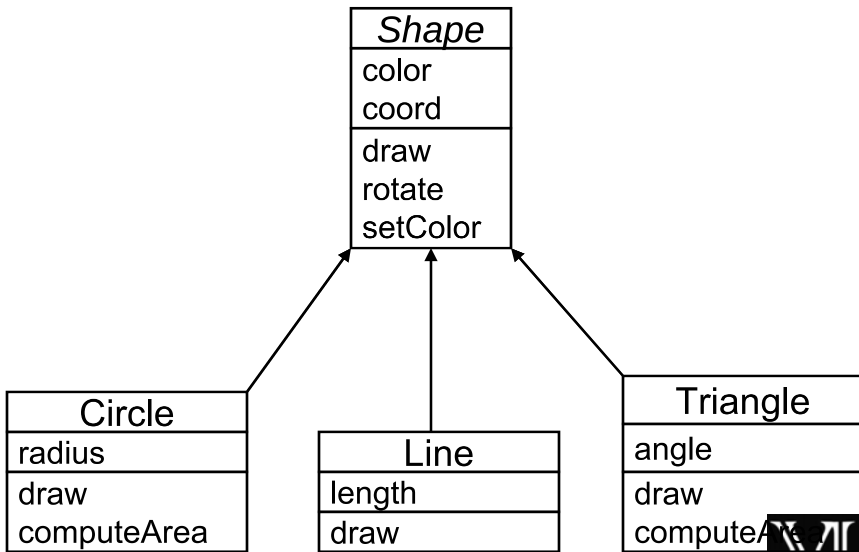
Inheritance – Advantages

- ►Reuse
- ►Less redundancy
- ►Increased maintainability

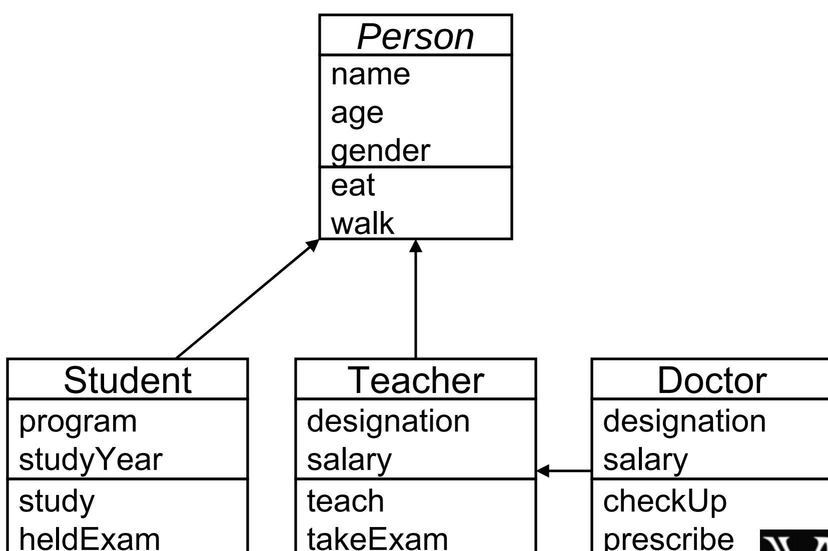
Reuse with Inheritance

- ►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

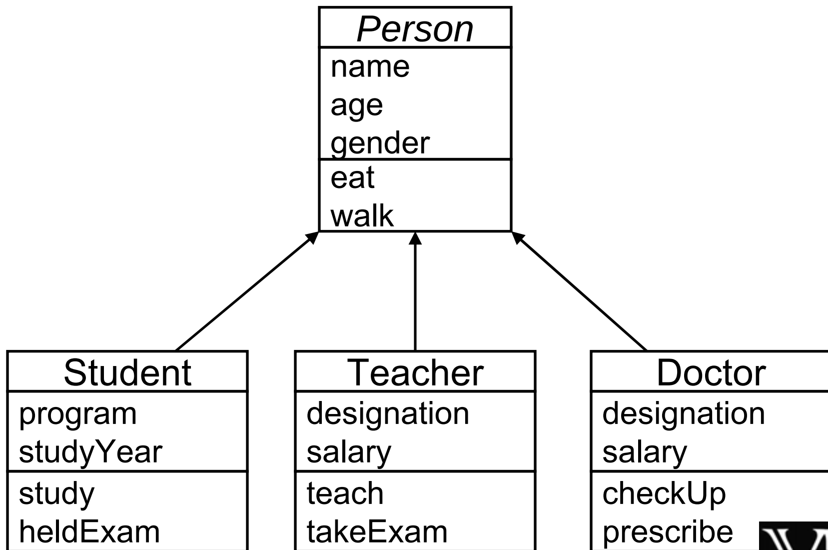
Example Reuse



Example Reuse



Example Reuse



Object-Oriented Programming (OOP)
Lecture No. 4

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

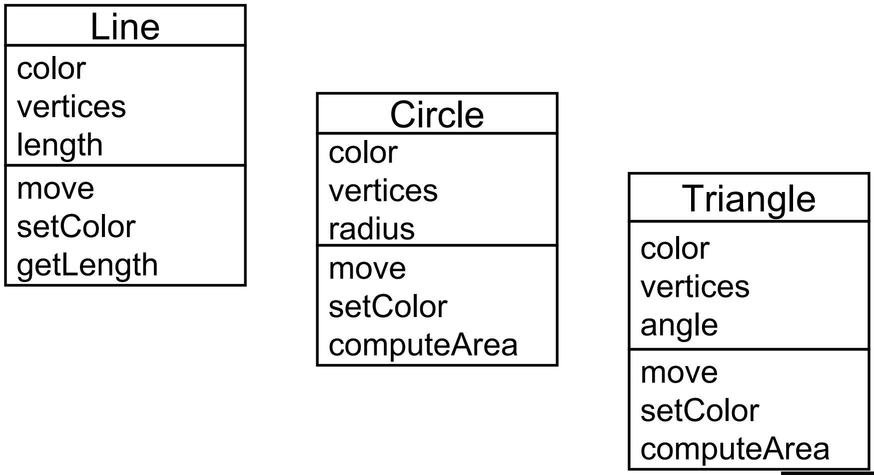
Concepts Related with Inheritance

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

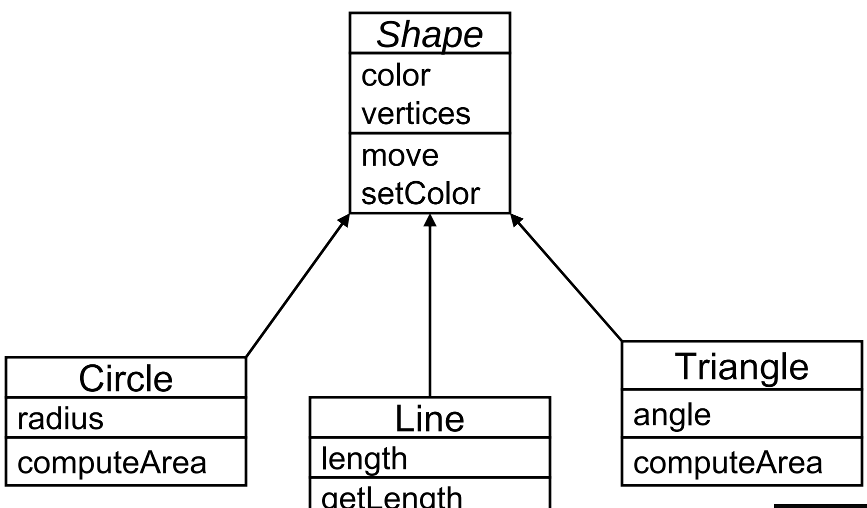
Generalization

- ► 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
- ► This concept is known as Generalization

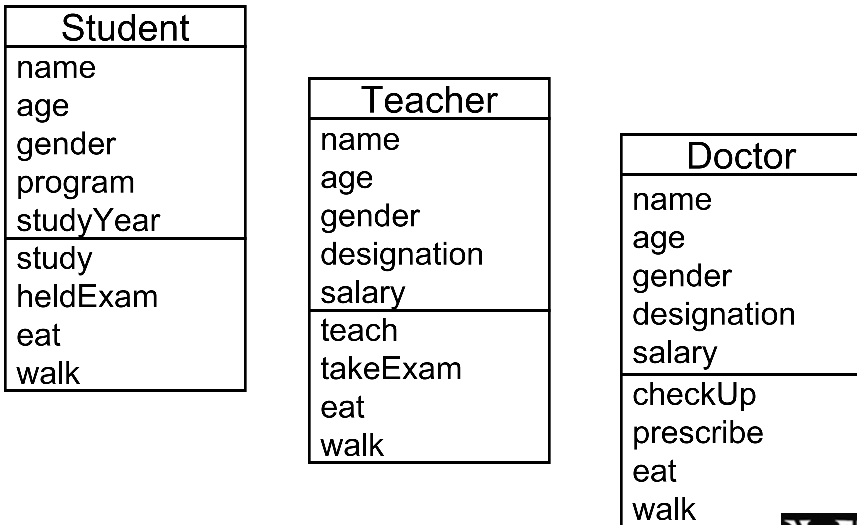
Example – Generalization



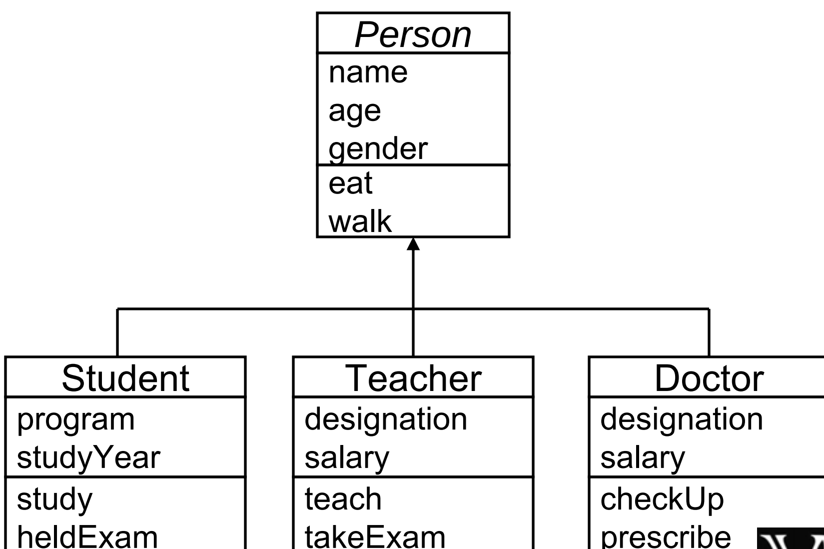
Example – Generalization



Example – Generalization



Example – Generalization



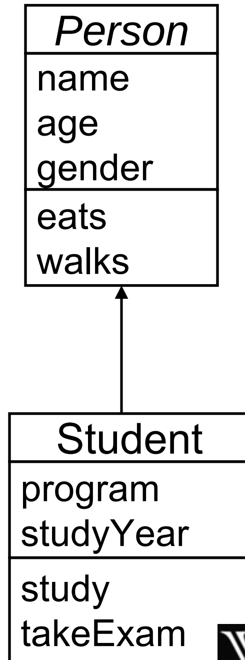
Sub-typing & Specialization

- We want to add a new class to an existing model
- 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

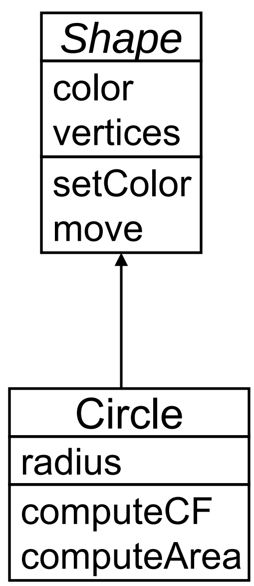
Sub-typing (Extension)

- Sub-typing means that derived class is behaviourally compatible with the base class
- Behaviourally compatible means that base class can be replaced by the derived class

Example –
Sub-typing
(Extension)



Example –
Sub-typing
(Extension)

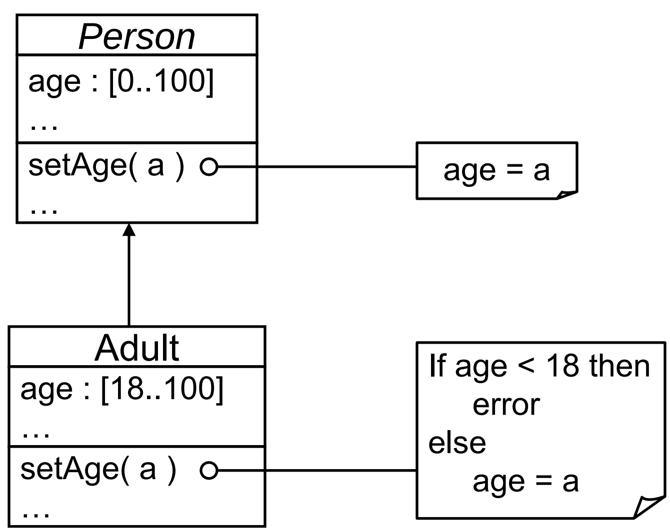


Specialization (Restriction)

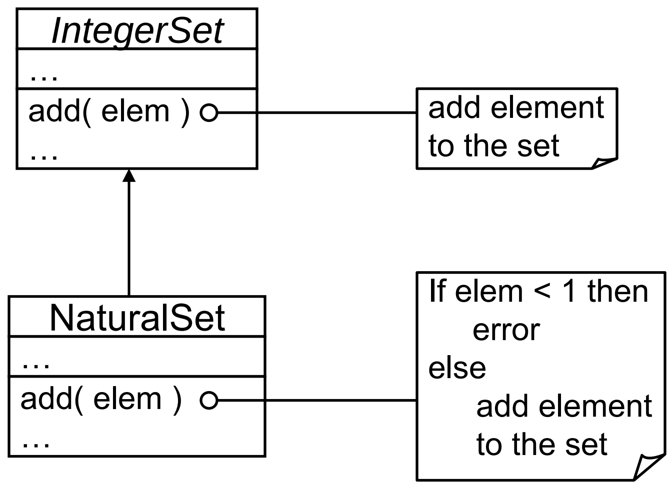
► Specialization means that derived class is behaviourally incompatible with the base class

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

Example – Specialization (Restriction)



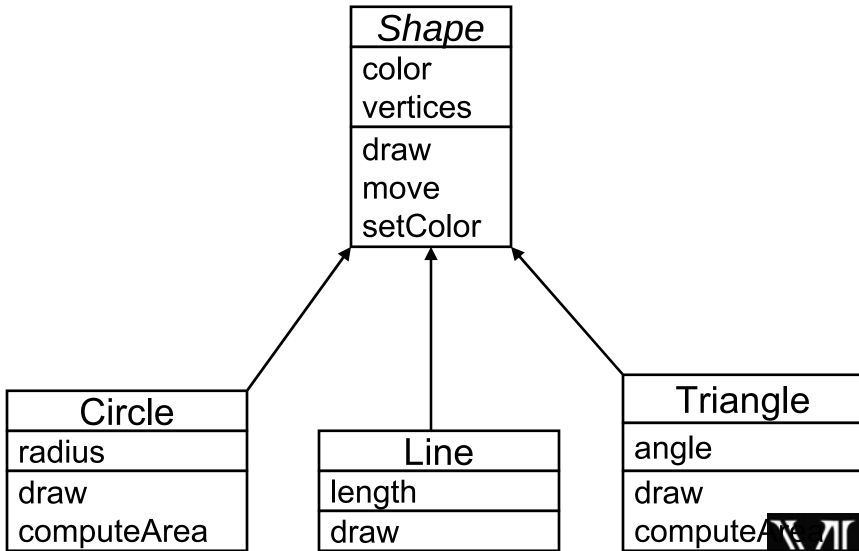
Example – Specialization (Restriction)



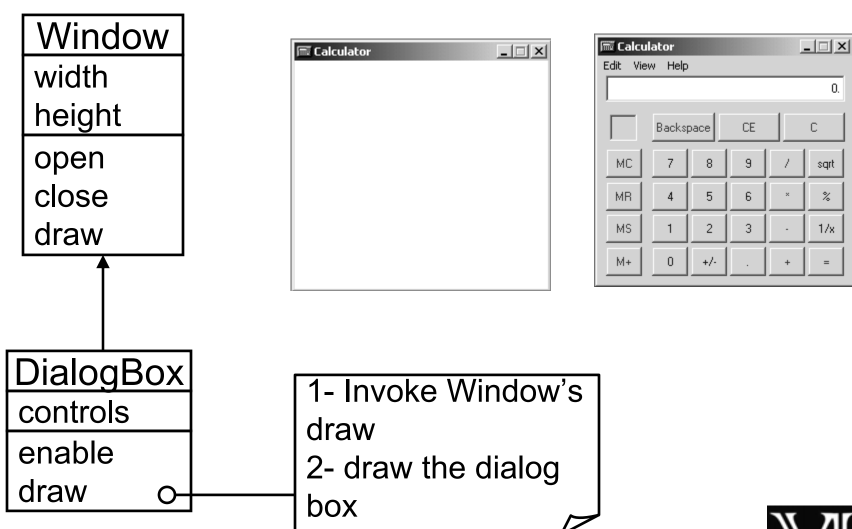
Overriding

- ►A class may need to override the default behaviour provided by its base class
- ►Reasons for overriding
  - Provide behaviour specific to a derived class
  - Extend the default behaviour
  - Restrict the default behaviour
  - Improve performance

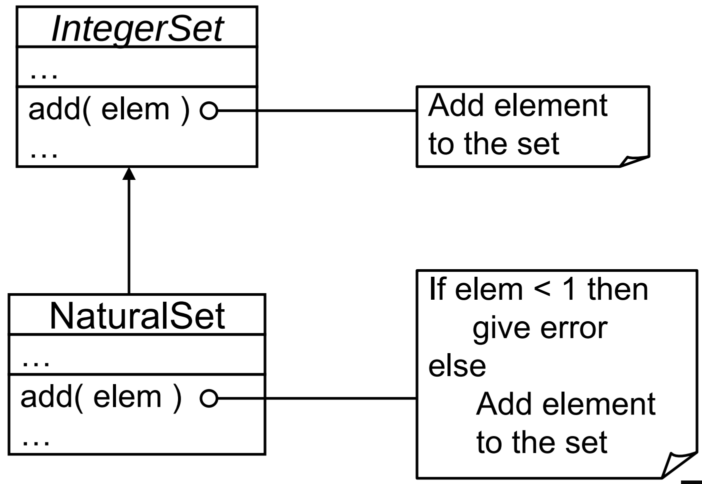
Example – Specific Behaviour



Example – Extension

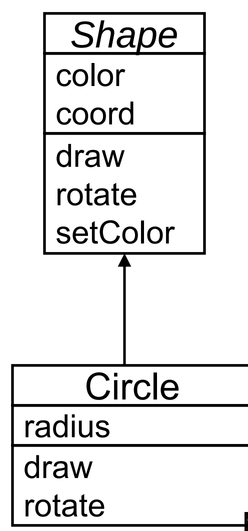


Example – Restriction



Example – Improve Performance

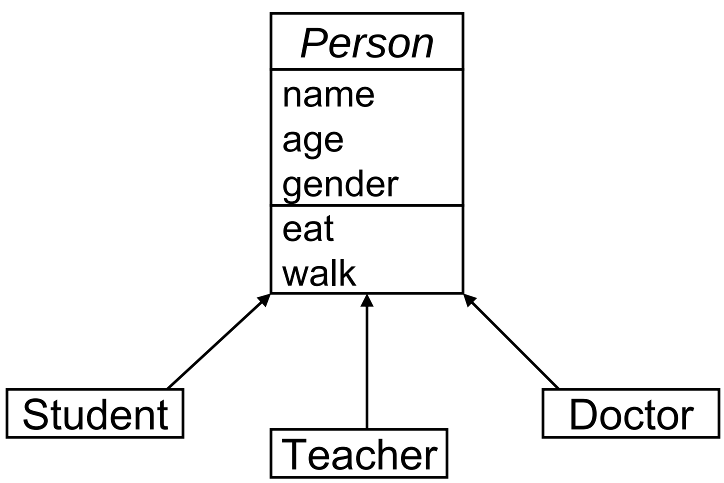
- Class Circle overrides *rotate* operation of class Shape with a Null operation.



Abstract Classes

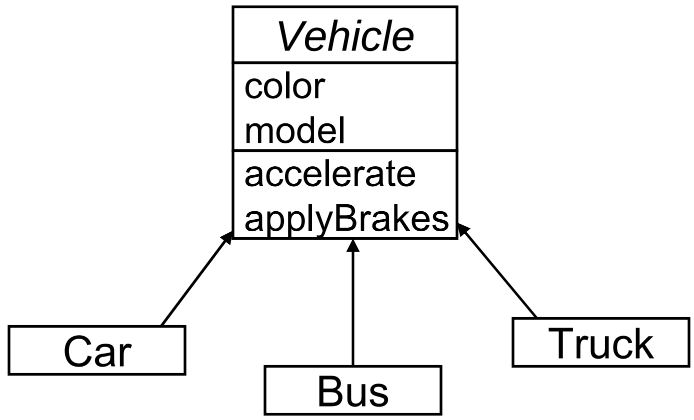
- ►An abstract class implements an abstract concept
- ►Main purpose is to be inherited by other classes
- ►Can't be instantiated
- ►Promotes reuse

Example – Abstract Classes



- ►Here, Person is an abstract class

Example – Abstract Classes



►Here, Vehicle is an abstract class

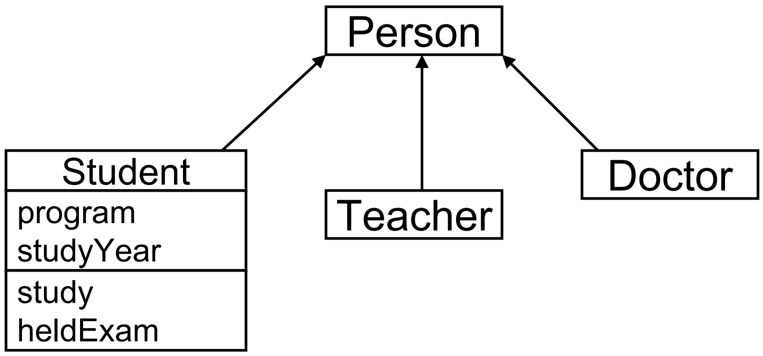
Concrete Classes

►A concrete class implements a concrete concept

►Main purpose is to be instantiated

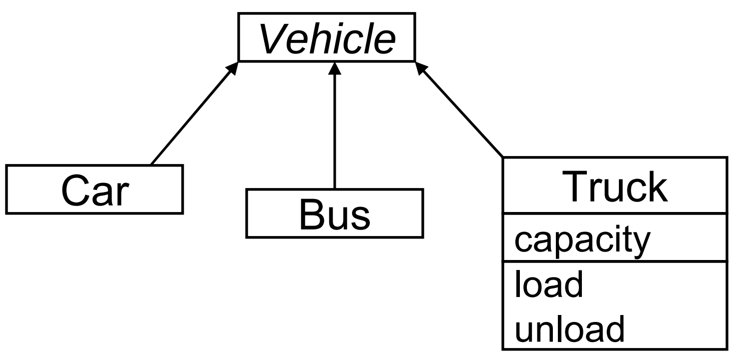
►Provides implementation details specific to the domain context

Example – Concrete Classes



► Here, Student, Teacher and Doctor are concrete classes

Example – Concrete Classes



- Here, Car, Bus and Truck are concrete classes

Object-Oriented Programming (OOP)
Lecture No. 5

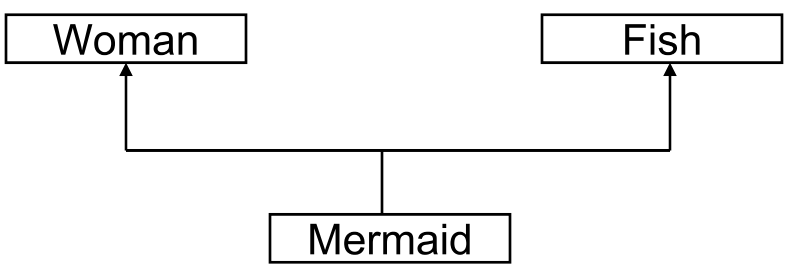
Multiple Inheritance

- We may want to reuse characteristics of more than one parent class

Example – Multiple Inheritance

Mermaid

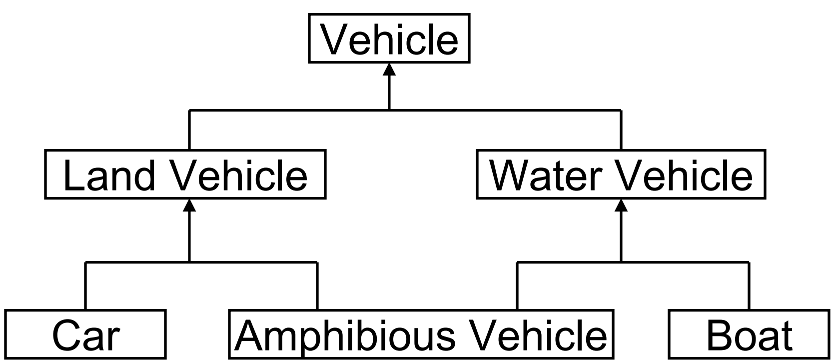
Example – Multiple Inheritance



Example – Multiple Inheritance

Amphibious Vehicle

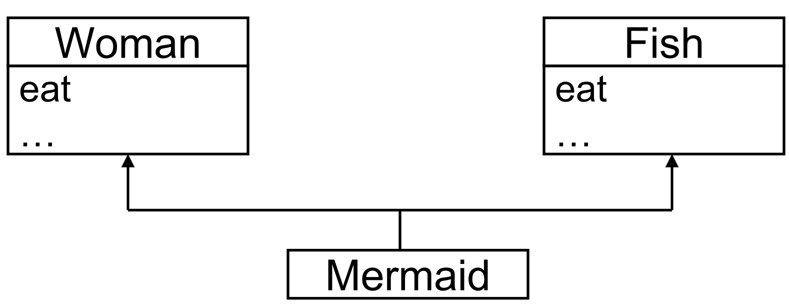
Example – Multiple Inheritance



Problems with Multiple Inheritance

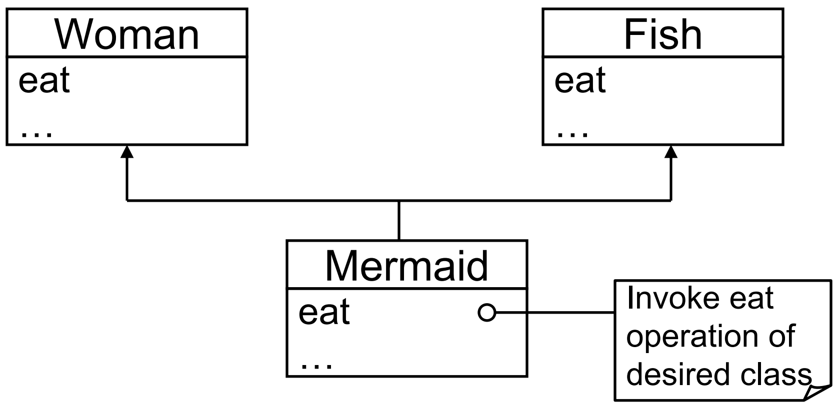
- ►Increased complexity
- ►Reduced understanding
- ►Duplicate features

Problem – Duplicate Features

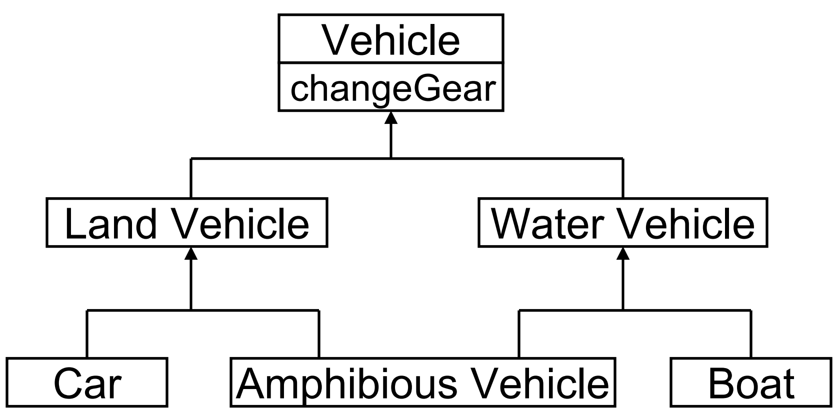


- ►Which *eat* operation *Mermaid* inherits?

Solution – Override the Common Feature



Problem – Duplicate Features (Diamond Problem)



- Which *changeGear* operation Amphibious Vehicle inherits?

Solution to Diamond Problem

- Some languages disallow diamond hierarchy
- Others provide mechanism to ignore characteristics from one side

Association

- Objects in an object model interact with each other
- Usually an object provides services to several other objects
- An object keeps associations with other objects to delegate tasks

Kinds of Association

- Class Association
  - Inheritance
- Object Association
  - Simple Association
  - Composition
  - Aggregation

Simple Association

- Is the weakest link between objects
- Is a reference by which one object can interact with some other object
- Is simply called as “association”

Kinds of Simple Association

- w.r.t navigation
  - One-way Association
  - Two-way Association
- w.r.t number of objects
  - Binary Association
  - Ternary Association
  - N-ary Association

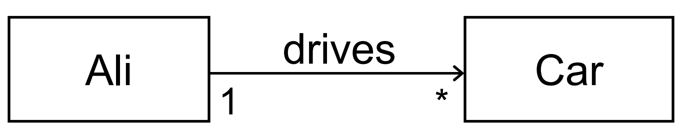
One-way Association

- We can navigate along a single direction only
- Denoted by an arrow towards the server object

Example – Association

► Ali lives in a House

Example – Association



► Ali drives his Car

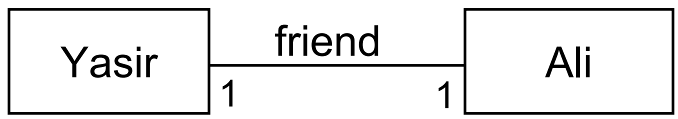
Two-way Association

- ►We can navigate in both directions
- ►Denoted by a line between the associated objects

Example – Two-way Association

- ►Employee works for company
- ►Company employs employees

Example – Two-way Association



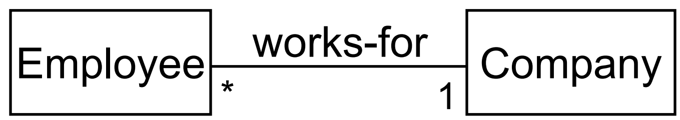
- Yasir is a friend of Ali
- Ali is a friend of Yasir

Binary Association

- Associates objects of exactly two classes

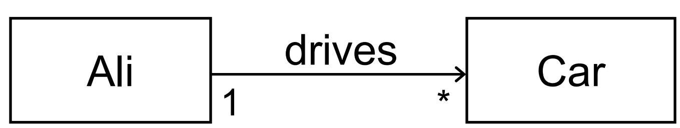
- Denoted by a line, or an arrow between the associated objects

Example – Binary Association



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

Example – Binary Association

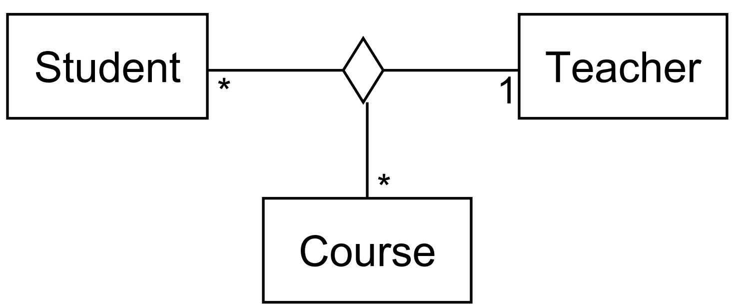


► Association "drives" associates objects of exactly two classes

Ternary Association

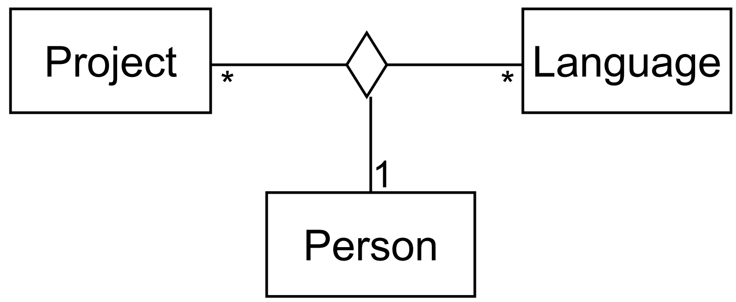
- ►Associates objects of exactly three classes
- ►Denoted by a diamond with lines connected to associated objects

Example – Ternary Association



- ►Objects of exactly three classes are associated

Example – Ternary Association



- ►Objects of exactly three classes are associated

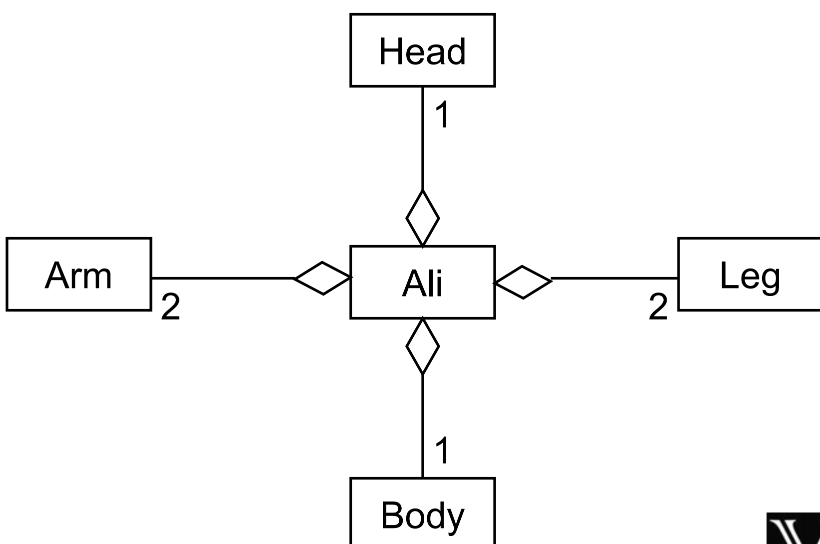
N-ary Association

- ►An association between 3 or more classes
- ►Practical examples are very rare

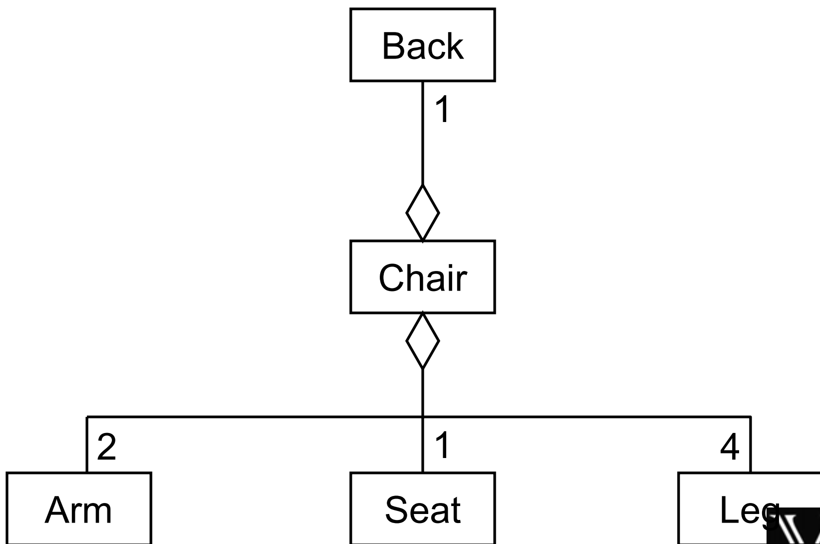
Composition

- 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



Example – Composition of Chair



Composition is Stronger

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

Example – Composition is Stronger

- ► Ali is made up of different body parts
- ► They can’t exist independent of Ali

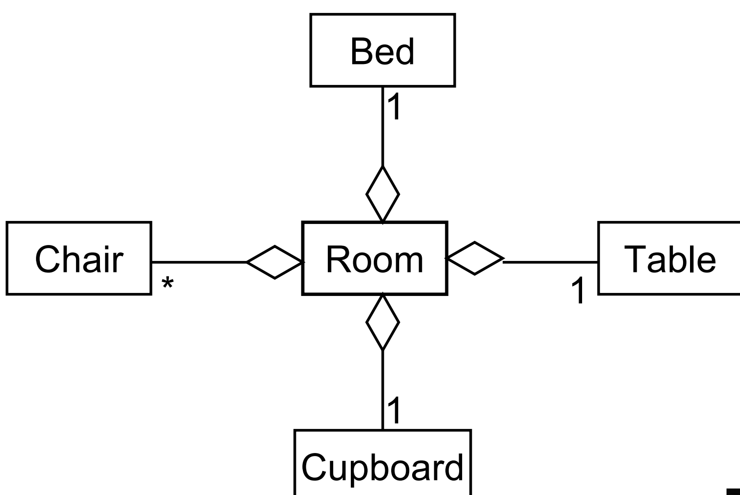
Example – Composition is Stronger

- ► Chair’s body is made up of different parts
- ► They can’t exist independently

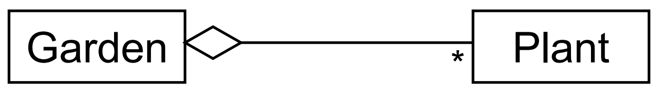
Aggregation

- 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

- Aggregation is weaker relationship, because
  - Aggregate object is not a part of the container
  - Aggregate object can exist independently

Example – Aggregation is Weaker

- ►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 – Aggregation is Weaker

- ►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

Object-Oriented Programming (OOP)
Lecture No. 6

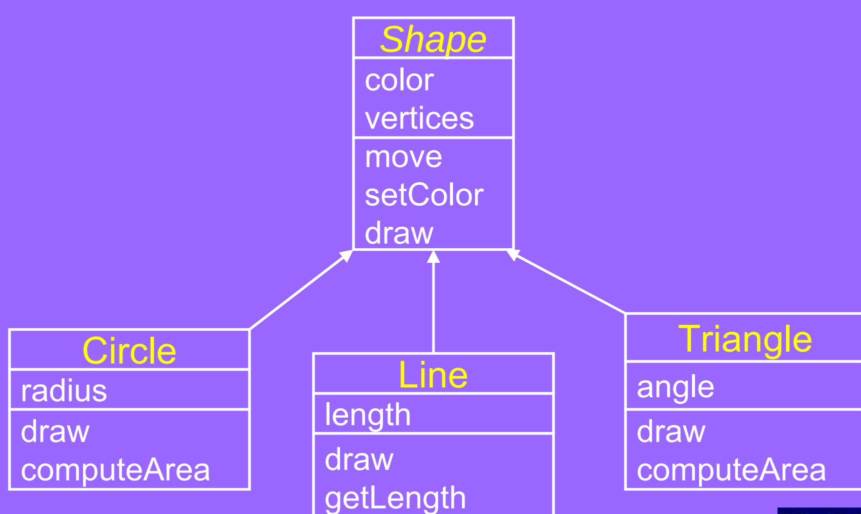
Class Compatibility

- 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

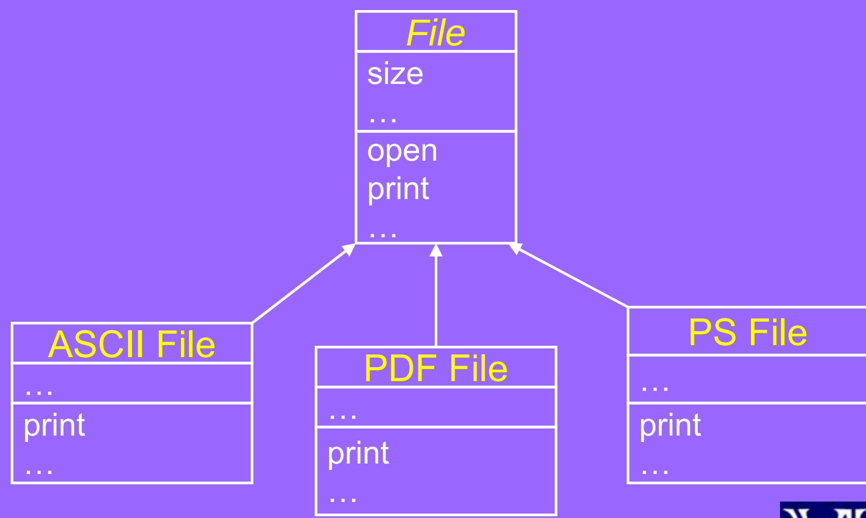
...Class Compatibility

- 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

Example – Class Compatibility



Example – Class Compatibility



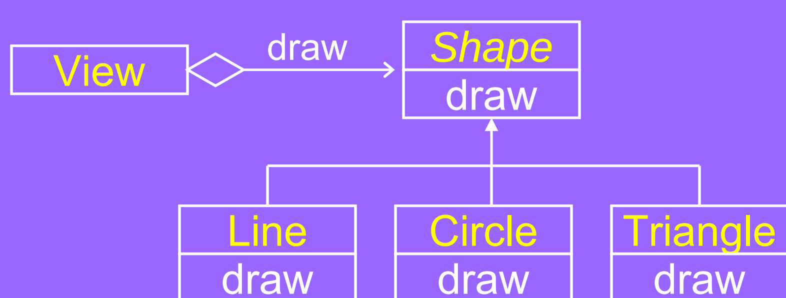
Polymorphism

- In general, polymorphism refers to existence of different forms of a single entity
- For example, both Diamond and Coal are different forms of Carbon

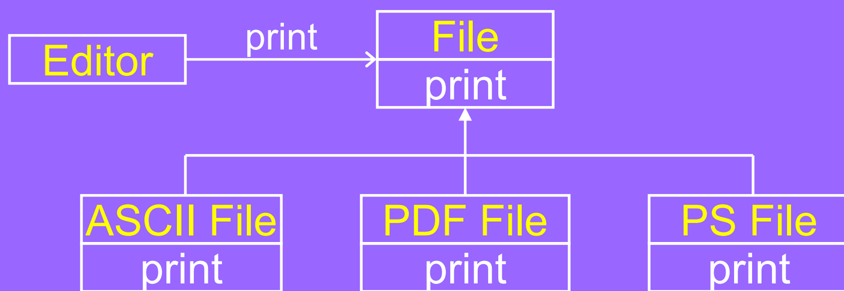
Polymorphism in OO Model

- 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

Example – Polymorphism

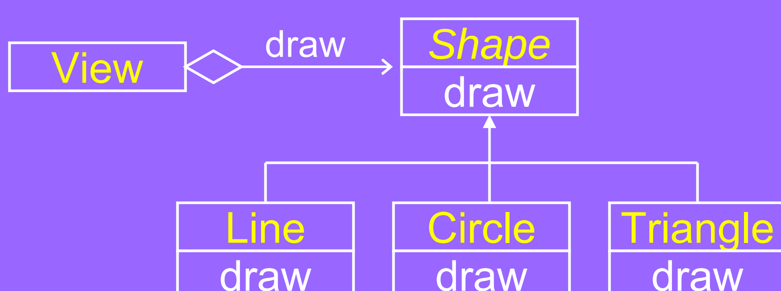


Example – Polymorphism



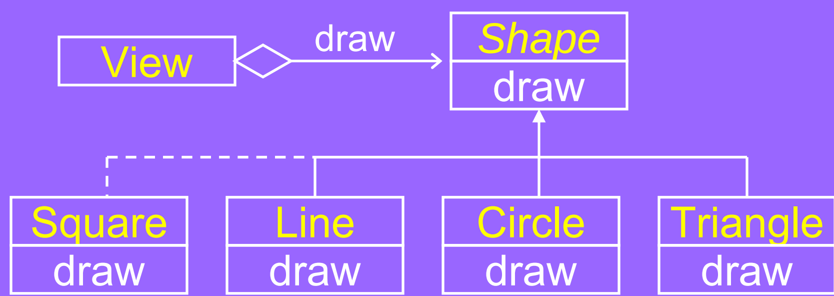
Polymorphism – Advantages

- Messages can be interpreted in different ways depending upon the receiver class



Polymorphism – Advantages

- New classes can be added without changing the existing model



Polymorphism – Advantages

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

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.

...Identify Classes

- Eliminate irrelevant classes

- Editor – Very broad scope

- User – Out of system boundary

...Identify Classes

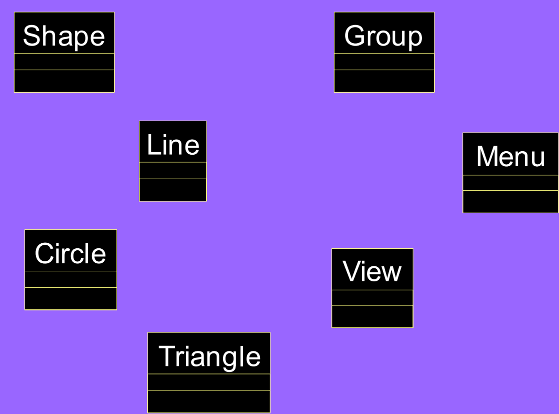
- Add classes by analyzing requirements
- Group – required to behave as a shape
  - "Individual shapes can be grouped together and can behave as a single shape"
- View – editor must have a display area

...Identify Classes

- Following classes have been identified:

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

Object Model – Graphic Editor



Identify Associations

- Extract verbs connecting objects

- "Individual shapes can be grouped together"
  - Group consists of lines, circles, triangles
  - Group can also consists of other groups

  (Composition)

... Identify Associations

- Verify access paths

- View contains shapes
  - View contains lines
  - View contains circles
  - View contains triangles
  - View contains groups

  (Aggregation)

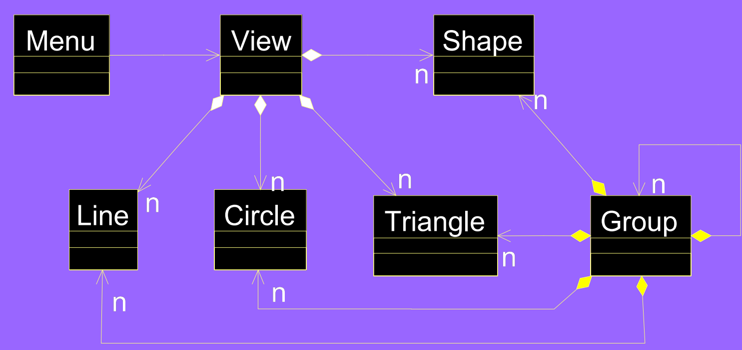
... Identify Associations

- Verify access paths

- Menu sends message to View

  (Simple One-Way Association)

Object Model – Graphic Editor



Identify Attributes

- Extract properties of the object
  - From the problem statement

- Properties are not mentioned

...Identify Attributes

- Extract properties of the object
  - From the domain knowledge

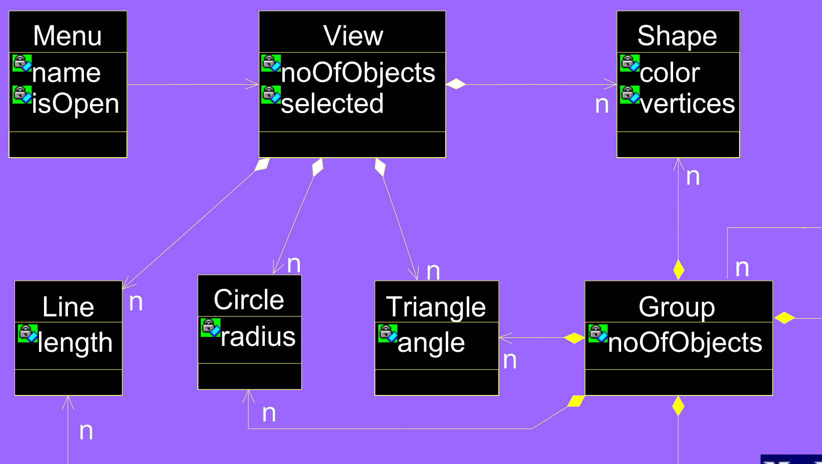
- Line
  - Color
  - Vertices
  - Length
- Circle
  - Color
  - Vertices
  - Radius
- Triangle
  - Color
  - Vertices
  - Angle
- Shape
  - Color
  - Vertices

...Identify Attributes

- Extract properties of the object
  - From the domain knowledge

- Group
  - noOfObjects
- View
  - noOfObjects
  - selected
- Menu
  - Name
  - isOpen

Object Model – Graphic Editor



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.

... Identify Operations

- ➢ Eliminate irrelevant operations
- ▶ Develop – out of system boundary
- ▶ Behave – have broad semantics

...Identify Operations

- ➢ Following are selected operations:

- Line
  - – Draw
  - – Select
  - – Move
  - – Rotate
- Circle
  - – Draw
  - – Select
  - – Move
  - – Rotate

...Identify Operations

➢ Following are selected operations:

- Triangle
  - Draw
  - Select
  - Move
  - Rotate
- Shape
  - Draw
  - Select
  - Move
  - Rotate

...Identify Operations

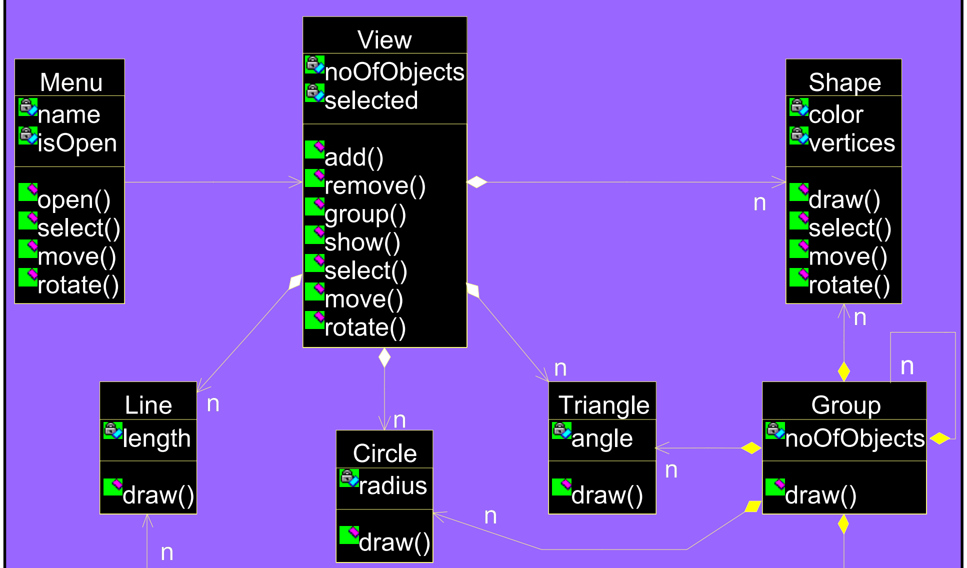
➢ Following are selected operations:

- Group
  - Draw
  - Select
  - Move
  - Rotate
- Menu
  - Open
  - Select
  - Move
  - Rotate

...Identify Operations

- Extract operations using domain knowledge

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



Identify Inheritance

- Search "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

…Identify Inheritance

- 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

...Refining the Object Model

- Share associations
- View contains all kind of shapes
- Group consists of all kind of shapes

...Refining the Object Model

- Share attributes

- Shape – Line, Circle, Triangle and Group
  - Color, vertices

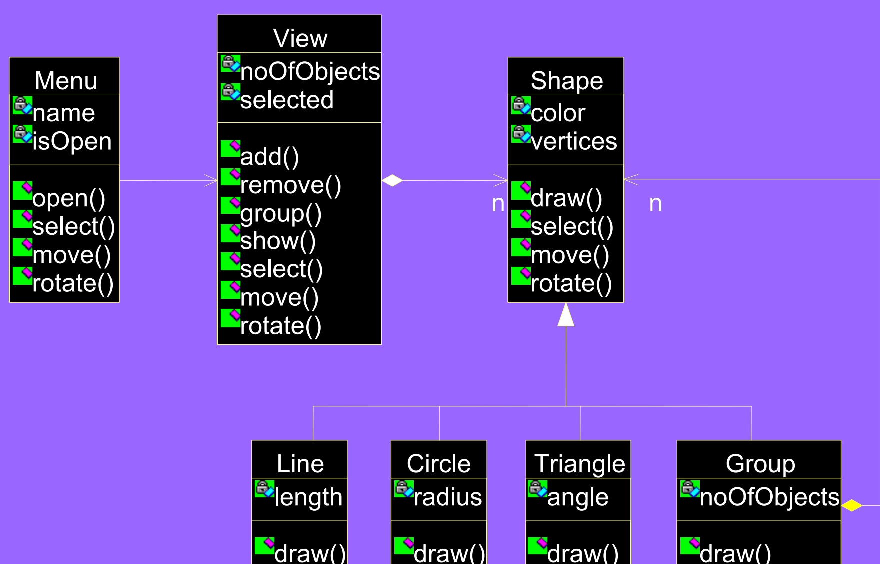
...Refining the Object Model

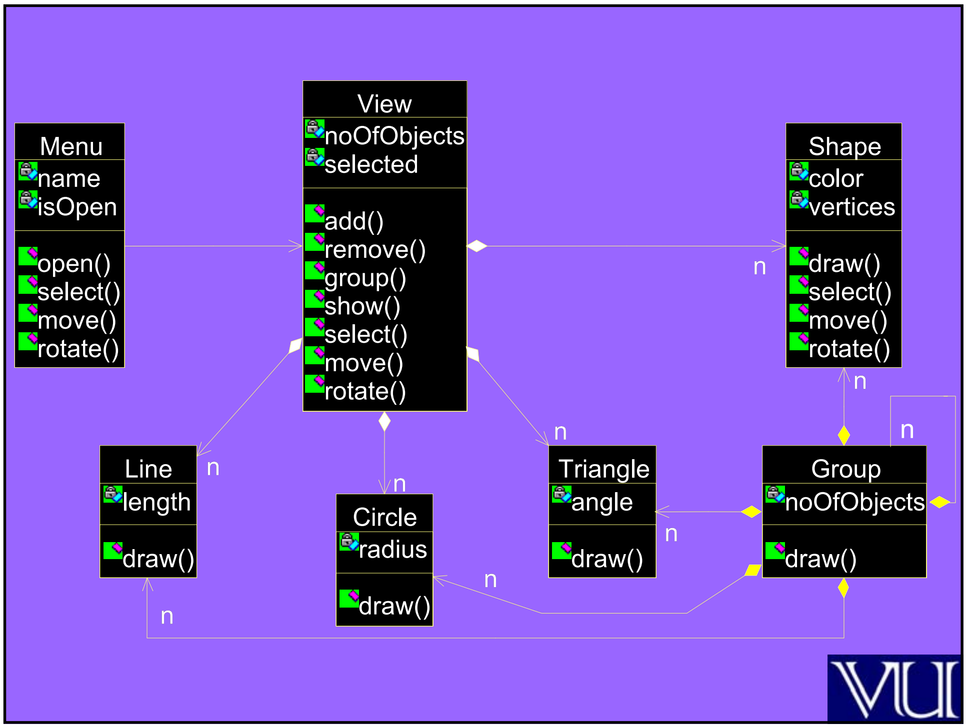
- Share operations

- Shape – Line, Circle, Triangle and Group
  - Select
  - Move
  - Rotate

...Refining the Object Model

- Share the interface and override implementation
- Shape – Line, Circle, Triangle and Group
  - Draw




View
noOfObjects
selected
add()
remove()
group()
show()
select()
move()
rotate()
Menu
name
isOpen
open()
select()
move()
rotate()
Shape
color
vertices
draw()
select()
move()
rotate()
Line
length
draw()
Circle
radius
draw()
Triangle
angle
draw()
Group
noOfObjects
draw()
n
n
n
n
n
n
n
n
n

Object oriented programming (OOP)
Lecture No. 7

Class

- ►Class is a tool to realize objects
- ►Class is a tool for defining a new type

Example

- Lion is an object
- Student is an object
- Both has some attributes and some behaviors

Uses

- The problem becomes easy to understand
- Interactions can be easily modeled

Type in C++

- ►Mechanism for user defined types are
  - Structures
  - Classes
- ►Built-in types are like int, float and double
- ►User defined type can be
  - Student in student management system
  - Circle in a drawing software

Abstraction

- ►Only include details in the system that are required for making a functional system
- ►Student
  - Name
  - Address
  - Sibling
  - Father Business

Relevant to our problem

Not relevant to our problem

Defining a New User Defined Type

```
class ClassName
{
  …
  DataType MemberVariable;
  ReturnType MemberFunction();
  …
};
```

Syntax

Syntax

Example

```
class Student
{
  int rollNo;
  char *name;
  float CGPA;
  char *address;
…
  void setName(char *newName);
  void setRollNo(int newRollNo);
…
};
```

Member variables

Member Functions

Why Member Function

- ►They model the behaviors of an object
- ►Objects can make their data invisible
- ►Object remains in consistent state

Example

```
Student aStudent;

aStudent.rollNo = 514;

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

Object and Class

► Object is an instantiation of a user defined type or a class

Declaring class variables

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

TypeName VaraibaleName;

int *var*;

Student aStudent;

Accessing members

- Members of an object can be accessed using
  - dot operator (.) to access via the variable name
  - arrow operator (->) to access via a pointer to an object
- Member variables and member functions are accessed in a similar fashion

Example

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

  Student aStudent;
  aStudent.rollNo;
```

Error

Access specifiers

Access specifiers

- There are three access specifiers
  - ‘public’ is used to tell that member can be accessed whenever you have access to the object
  - ‘private’ is used to tell that member can only be accessed from a member function
  - ‘protected’ to be discussed when we cover inheritance

Example

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

Cannot be accessed outside class

Can be accessed outside class

Example

```
class Student{
...
  int rollNo;
public:
  void setRollNo(int aNo);
};
int main(){
  Student aStudent;
  aStudent.SetRollNo(1);
}
```

Default access specifiers

- When no access specifier is mentioned then by default the member is considered private member

Example

```
class Student
{
  char * name;
  int RollNo;
};
```

```
class Student
{
private:
  char * name;
  int RollNo;
};
```

Example

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

Error

Example

```
class Student
{
  char * name;
  int RollNo;
public:
  void setName(char *);
};
Student aStudent;
aStudent.SetName(“Ali”);
```

Object Oriented Programming
(OOP)
Lecture No. 8

Review

- Class
  - Concept
  - Definition
- Data members
- Member Functions
- Access specifier

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

Member Functions (contd.)

- Define member function inside the class definition

OR

- Define member function outside the class definition
  - But they must be declared inside class definition

Function Inside Class Body

```
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

Example

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

Function Outside Class Body

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

Scope resolution operator

Example

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

Inline Functions

- Instead of calling an inline function compiler replaces the code at the function call point
- Keyword ‘inline’ is used to request compiler to make a function inline
- It is a request and not a command

Example

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

Inline Functions

- If we define the function inside the class body then the function is by default an inline function
- In case function is defined outside the class body then we must use the keyword ‘inline’ to make a function inline

Example

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

Example

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

Example

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

Example

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

Constructor

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
- ►Constructor is automatically called when the object is created
- ►Constructor are not usually called explicitly

Constructor (contd.)

- 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{
  …
public:
  Student(){
     rollNo = 0;
     …
  }
};
```

Example

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

Default Constructor

- Constructor without any argument is called default constructor
- If we do not define a default constructor the compiler will generate a default constructor
- This compiler generated default constructor initialize the data members to their default values

Example

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

Example

Compiler generated default constructor

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

Constructor Overloading

- ►Constructors can have parameters
- ►These parameters are used to initialize the data members with user supplied data

Example

```
class Student{
…
public:
  Student();
  Student(char * aName);
  Student(char * aName, int aRollNo);
  Student(int aRollNo, int aRollNo,
  float aGPA);
};
```

Example

```
Student::Student(int aRollNo,
               char * aName){
  if(aRollNo < 0){
     rollNo = 0;
  }
  else {
  rollNo = aRollNo;
  }
  …
}
```

Example

```
int main()
{
  Student student1;
  Student student2(“Name”);
  Student student3(”Name”, 1);
  Student student4(”Name”,1,4.0);
}
```

Constructor Overloading

- ►Use default parameter value to reduce the writing effort

Example

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

Is equivalent to

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

Copy Constructor

- ►Copy constructor are used when:
  - Initializing an object at the time of creation
  - When an object is passed by value to a function

Example

```
void func1(Student student){
…
}
int main(){
 Student studentA;
 Student studentB = studentA;
 func1(studentA);
}
```

Copy Constructor (Syntax)

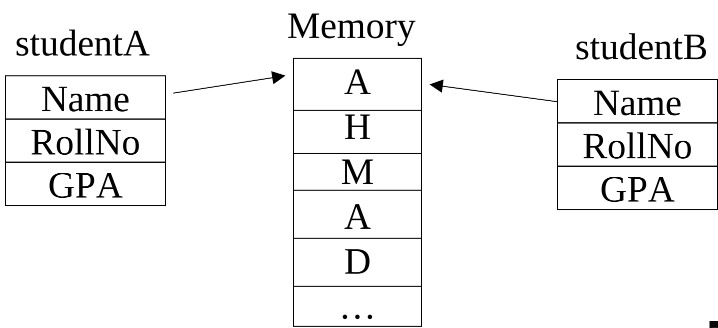
```
Student::Student(
        const Student &obj){
  rollNo = obj.rollNo;
  name = obj.name;
  GPA = obj.GPA;
}
```

Shallow Copy

- When we initialize one object with another then the compiler copies state of one object to the other
- This kind of copying is called shallow copying

Example

Student studentA;
Student studentB = studentA;



Copy Constructor (contd.)

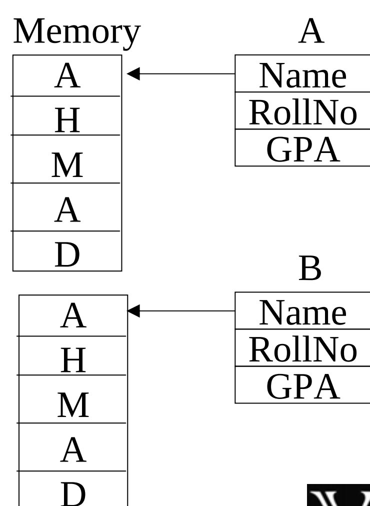
```
Student::Student(
        const Student & obj){
  int len = strlen(obj.name);
  name = new char[len+1]
  strcpy(name, obj.name);
  …
  //copy rest of the data members
}
```

Copy Constructor (contd.)

- ►Copy constructor is normally used to perform deep copy
- ►If we do not make a copy constructor then the compiler performs shallow copy

Example

Student studentA;
Student studentB = studentA;



Object Oriented Programming
(OOP)
Lecture No. 9

Review

► Member functions implementation
► Constructors
► Constructors overloading
► Copy constructors

Copy Constructor

► Copy constructor are used when:
  - Initializing an object at the time of creation
  - When an object is passed by value to a function

Example

```
void func1(Student student){
…
}
int main(){
 Student studentA("Ahmad");
 Student studentB = studentA;
 func1(studentA);
}
```

Copy Constructor (contd.)

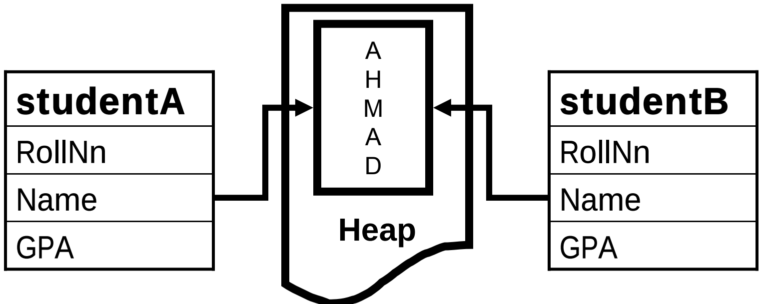
```
Student::Student(
   const Student &obj){
 rollNo = obj.rollNo;
 name = obj.name;
 GPA = obj.GPA;
}
```

Shallow Copy

- When we initialize one object with another then the compiler copies state of one object to the other
- This kind of copying is called shallow copying

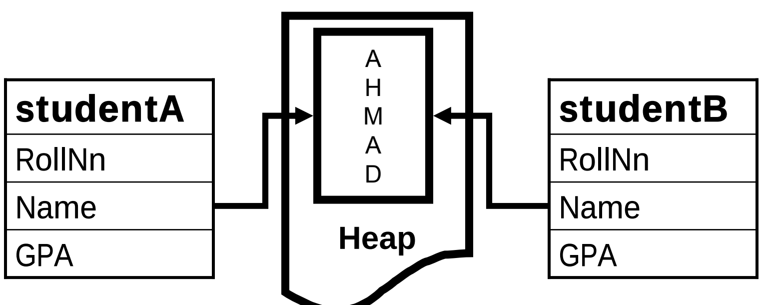
Example

Student studentA(“Ahmad”);
Student studentB = studentA;



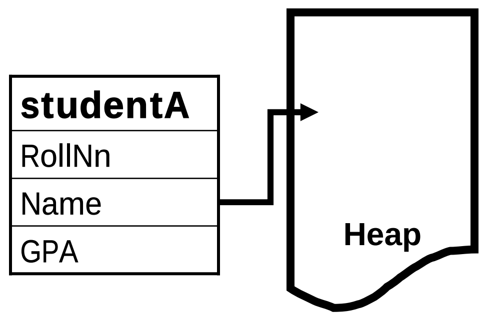
Example

```
int main(){
 Student studentA(“Ahmad”,1);
 {
  Student studentB = studentA;
```



Example

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

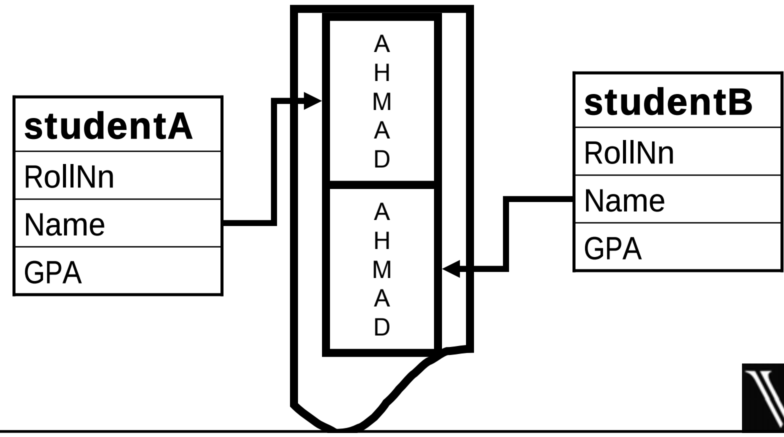


Copy Constructor (contd.)

```
Student::Student(
            const Student & obj){
  int len = strlen(obj.name);
  name = new char[len+1]
  strcpy(name, obj.name);
  …
  /*copy rest of the data members*/
}
```

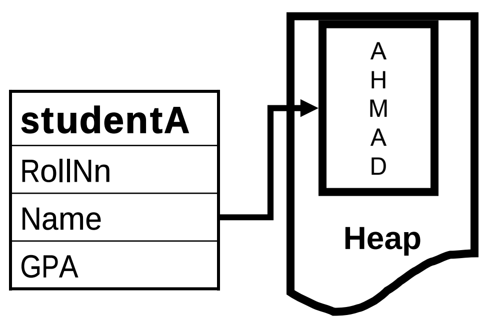
Example

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



Example

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



Copy Constructor (contd.)

- ►Copy constructor is normally used to perform deep copy
- ►If we do not make a copy constructor then the compiler performs shallow copy

Destructor

- ►Destructor is used to free memory that is allocated through dynamic allocation
- ►Destructor is used to perform house keeping operations

Destructor (contd.)

- 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 << “Cons\n”;
 }
 Student::~Student(){
   cout << name << “Dest\n”;
 }
};
```

Example

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

Example

Output:

Ali Cons

Ahmad Cons

Ahmad Dest

Ali Dest

Accessor Functions

- Usually the data member are defined in private part of a class – information hiding
- Accessor functions are functions that are used to access these private data members
- Accessor functions also useful in reducing error

Example – Accessing Data Member

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

Example – Avoiding Error

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

Example - Getter

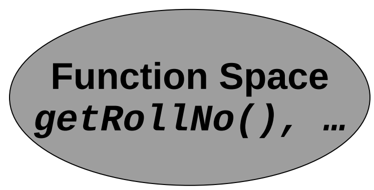
```
class Student{
 …
 int rollNo;
public:
 int getRollNo(){
   return rollNo;
 }
};
```

this Pointer

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

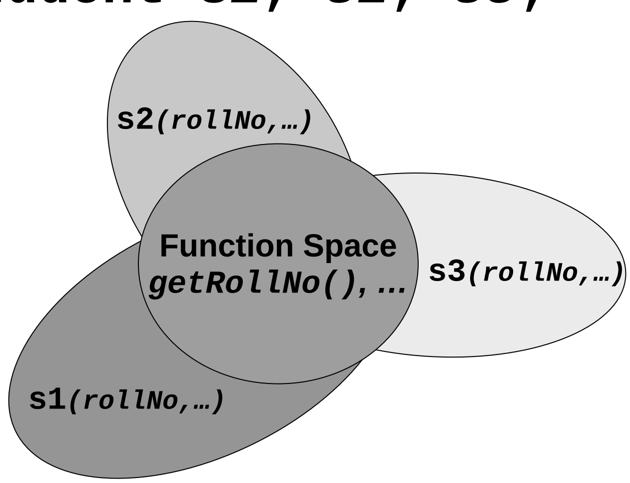
this Pointer

- 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*;**

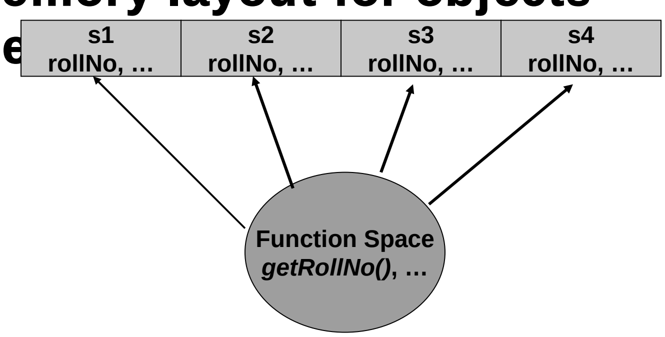


this Pointer

- ►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

this Pointer

- ►Memory layout for objects cre



•How does the functions know on which object to act?

this Pointer

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

s1 rollNo, …	s2 rollNo, …	s3 rollNo, …	s4 rollNo, …
address	*address*	*address*	*address*

•The variable containing the "self-address" is called this pointer

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

```
Student::Student(){
 rollNo = 0;
}

Student::Student(){
 this->rollNo = 0;
}
```

Object Oriented Programming
(OOP)
Lecture No. 10

Review

- ►Copy constructors
- ►Destructor
- ►Accessor Functions
- ►this Pointer

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;
}
```

Example

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

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

  return 0;
}
```

Separation of interface and implementation

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

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

Old implementation

Complex
float x
float y
float getX()
float getY()
void setNumber (float i, float j)
…

New implementation

Complex
float z
float theta
float getX()
float getY()
void setNumber (float i, float j)
…

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

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

Separation of interface and implementation

- ►Usually functions are defined in implementation files (.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(){
…
}
```

Driver.cpp

```
#include "student.h"

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

`const` Member Functions

- There are functions that are meant to be read only
- There must exist a mechanism to detect error if such functions accidentally change the data member

`const` Member Functions

► Keyword **`const`** is placed at the end of the parameter list

`const` Member Functions

Declaration:

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

Definition:

```
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
- They are just ***“read-only”***
- Errors due to typing are also caught at compile time

Example

```
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;
}
```

Example

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

const Functions

- Constructors and Destructors cannot be **const**
- Constructor and destructor are used to modify the object to a well defined state

Example

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

const Function

- ►Constant member function cannot change data member
- ►Constant member function cannot access non-constant member functions

Example

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

this Pointer and const Member Function

► this pointer is passed as constant pointer to const data in case of constant member functions

`const Student *const this;`

`instead of`

`Student * const this;`

Object Oriented Programming
(OOP)
Lecture No. 11

Review

- ► this Pointer
- ► Separation of interface and implementation
- ► Constant member functions

Problem

- Change the class Student such that a student is given a roll number when the object is created and cannot be changed afterwards

Student Class

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

Modified Student Class

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

Example

```
Student::Student(int aRollNo)
{
  rollNo = aRollNo;
  /*error: cannot modify a
  constant data member*/
}
```

Example

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

Member Initializer List

- A member initializer list is a mechanism to initialize data members
- It is given after closing parenthesis of parameter list of constructor
- In case of more then 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){
     …
  }
…
};
```

Order of Initialization

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

Example

```
class ABC{
 int x;
 int y;
 int z;
public:
 ABC();
};
```

Example

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

`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

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

Example

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

`const` Objects

- ►**`const`** objects cannot access "*non const*" member function
- ►Chances of unintentional modification are eliminated

Example

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

Example

```
int main(){
  const Student aStudent;
  int a = aStudent.getRollNo();
}
```

Constant data members

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

Static Variables

- 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

```
void func1(int i){
  static int staticInt = i;
  cout << staticInt << endl;
}
int main(){
  func1(1);
  func1(2);
}
```

Output:
1
1

Static Data Member

Definition

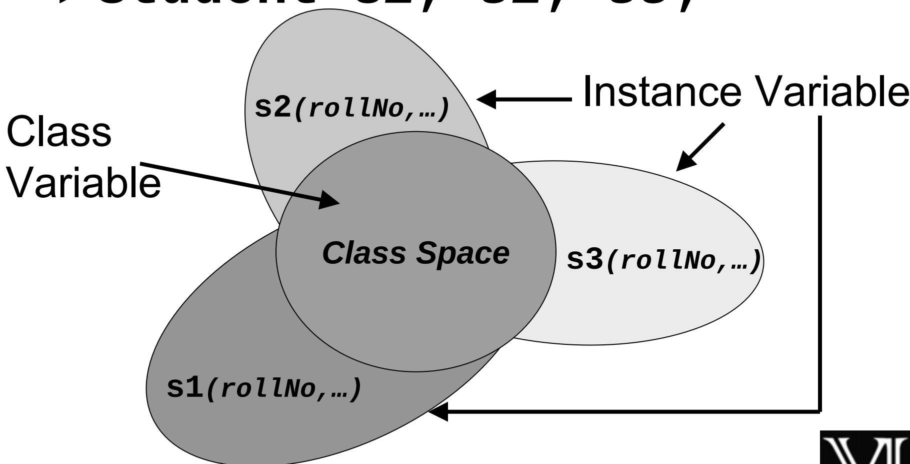
"A variable that is part of a class, yet is not part of an object of that class, is called static data member"

Static Data Member

- ►They are shared by all instances of the class
- ►They do not belong to any particular instance of a class

Class vs. Instance Variable

► **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

- ►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;
```

Object Oriented Programming
(OOP)
Lecture No. 12

Review

- ►Constant data members
- ►Constant objects
- ►Static data members

Static Data Member

Definition

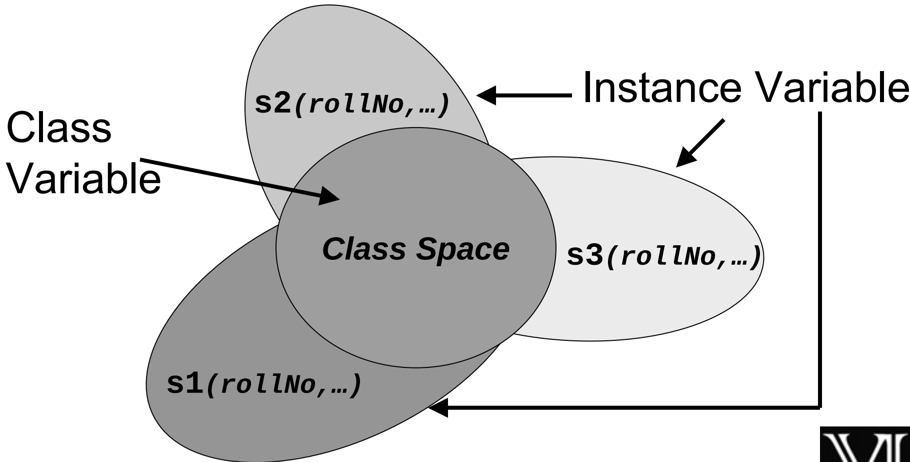
"A variable that is part of a class, yet is not part of an object of that class, is called static data member"

Static Data Member

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

Class vs. Instance Variable

► **`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

- ►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;
```

Accessing Static Data Member

- To access a static data member there are two ways
  - Access like a normal data member
  - 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;
}
```

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;
}
```

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

Example

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

Example

```
Student::Student(){
   noOfStudents++;
}
Student::~Student(){
   noOfStudents--;
}
```

Example

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

Output:
0
1
2

Problem

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

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”

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();
}
```

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*/
}
```

this Pointer

- *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

Global Variable vs. Static Members

- ►Alternative to static member is to use global variable
- ►Global variables are accessible to all entities of the program
  - Against information hiding

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
}
```

Example

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

Example

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

Example

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

Example

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

Object Oriented Programming
(OOP)
Lecture No. 13

Review

- ►Static data members
- ►Static member functions
- ►Array of objects

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:
  Studen();
  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;
}
```

Breakup of new Operation

- new operator is decomposed as follows
  - Allocating space in memory
  - Calling the appropriate constructor

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;
     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;
  }
  …
}
```

Recap

Object-Oriented Programming (OOP)
Lecture No. 14

Composition

Consider the following implementation of the student class:

Student
gpa : float rollNo : int name : char *
Student(char * = NULL, int = 0, float = 0.0); Student(const Student &) GetName() const : const char * SetName(char *) : void ~Student() …

Composition

```
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;
  ~Student();
  …
};
```

Composition

```
Student::Student(char * _name, int roll,
                              float g){
    cout << "Constructor::Student..\n";
    if (!_name){
          name = new char[strlen(_name)+1];
          strcpy(name,_name);
    }
    else name = NULL;
    rollNumber = roll;
    gpa = g;
}
```

Composition

```
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;
}
```

Composition

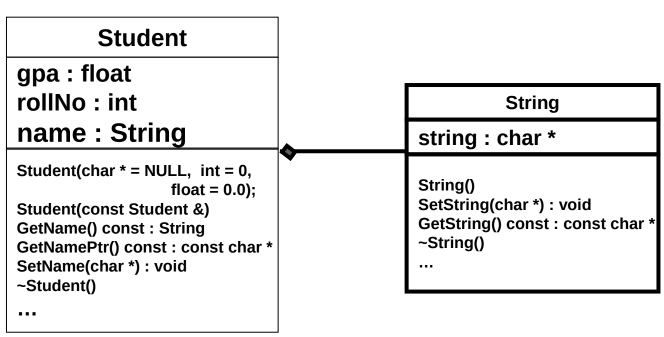
```
const char * Student::GetName(){
   return name;
}

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

Composition

- ►C++: "*its all about code reuse*"
- ►Composition:
  - ▪Creating objects of one class inside another class
- ►***"Has a"*** relationship:
  - ▪Bird has a beak
  - ▪Student has a name

Composition

Conceptual notation:



Composition

```
class String{
  private:
     char * ptr;
  public:
     String();
     String(const String &);
     void SetString(char *);
     const char * GetString() const;
     ~String()
     …
};
```

Composition

```
  String::String(){
     cout << "Constructor::String..\n";
     ptr = NULL;
  }

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

Composition

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

Composition

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

Composition

```
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();
  …
};
```

Composition

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

Composition

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

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

Composition

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

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

Composition

Main Function:

```
void main(){
      Student aStudent("Fakhir", 899,
                                  3.1);
      cout << endl;
      cout << “Name:”
            << aStudent.GetNamePtr()
            << “\n”;
}
```

Composition

► Output:

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

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

Composition

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

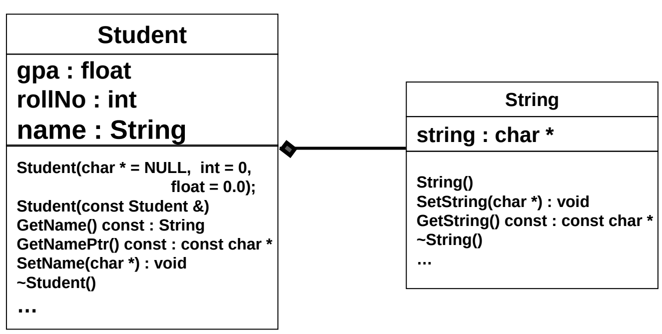
► Example:

- Constructor for the sub-object **`name`** is executed before the constructor of **`Student`**

Object-Oriented Programming (OOP)
Lecture No. 15

Composition

Conceptual notation:



Composition

Main Function:

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

Composition

► Output:

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

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

Composition

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

Composition

- ►To assign meaningful values to the object, the function **`SetString`** is called explicitly in the constructor
- ►This is an overhead
- ►Sub-object **`name`** in the **`student`** class can be initialized using the constructor
- ►"*Member initialization list*" syntax is used

Composition

►Add an overloaded constructor to the **`String`** class defined above:

```
class String{
    char *ptr;
public:
    String();
    String(char *);   //String(char * = NULL);
    String(const String &);
    void SetName(char *);
    ~String();
    …
};
```

Composition

```
String::String(char * str){
    if(str != NULL){
        ptr = new char[strlen(str)+1];
        strcpy(ptr, str);
    }
    else ptr = NULL;
    cout << "Overloaded
        Constructor::String..\n";
}
```

Composition

- ►Student class is modified as follows:

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

Composition

- ►Student class continued:

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

Composition

Main Function:

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

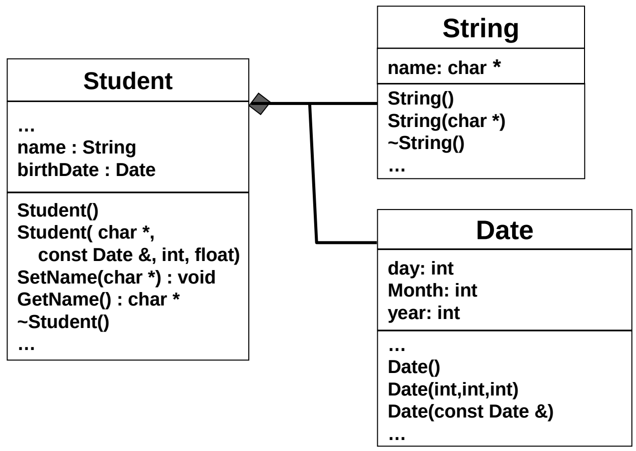
Composition

► Output:

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

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

Composition

Now consider the following case:



Composition

► Student class is modified as follows:

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

Composition

► Student class continued:

```
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";
}
```

Composition

► Main function:

```
int main(){
  Date _date(31, 12, 1982);
  Student aStudent("Fakhir",
                        _date,899,3.5);
  return 0;
}
```

Composition

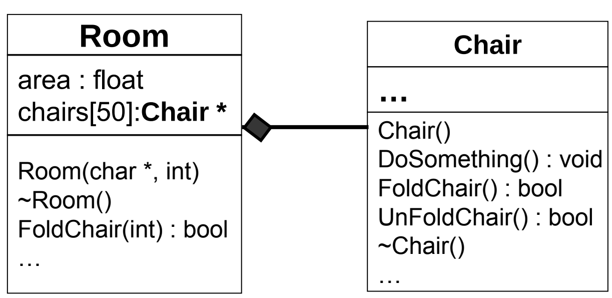
► Output:

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

Aggregation

Composition vs. Aggregation

► Aggregation is a *weak relationship*



Aggregation

- ►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);
   …
};
```

Aggregation

```
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;
}
```

Aggregation

```
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;
}
```

Aggregation

```
int main(){
   Chair ch1;
   {
      Room r1;
      r1.AddChair(&ch1, 1);
      r1.FoldChair(1);
   }
   ch1.UnFoldChair(1);
   return 0;
}
```

Friend Functions

► Consider the following class:

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

Friend Functions

► Global function:

```
void DoSomething(X obj){
    obj.a = 3; //Error
    obj.b = 4; //Error
}
```

Friend Functions

► In order to access the member variables of the class, function definition must be made a friend function:

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

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

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

Friend Classes

•Example:

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

Friend Classes

```
class Y{
private:
     int y_var1, y_var2;
     X objX;
public:
     void setX(){
          objX.x_var1 = 1;
     }
};
```

Friend Classes

```
int main(){
     Y objY;
     objY.setX();
     return 0;
}
```

Object-Oriented Programming (OOP)
Lecture No. 16

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 &);
    …
}
```

Operator overloading

► **Function implementation:**

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

Operator overloading

► The following statement:

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

Adds the contents of **c2** to **c1** and assigns it to **c3** (copy constructor)

Operator overloading

- ►To perform operations in a single mathematical statement e.g:

```
c1+c2+c3+c4
```

- ►We have to explicitly write:

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

Operator overloading

- ►Alternative way is:

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

Operator overloading

►If the mathematical expression is big:

- Converting it to C++ code will involve complicated mixture of function calls
- Less readable
- Chances of human mistakes are very high
- Code produced is very hard to maintain

Operator overloading

►C++ provides a very elegant solution:

“***Operator overloading***”

►C++ allows you to overload common operators like **+**, **-** or ***** etc...

►Mathematical statements don’t have to be explicitly converted into function calls

Operator overloading

- ►Assume that operator **+ has** been overloaded
- ►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
- ►Example of predefined types:

```
int
float
double
char
long
```

Operator overloading

► Example:

```
float x;
int y;
x = 102.02 + 0.09;
Y = 50 + 47;
```

Operator overloading

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

Operator overloading

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

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

Operator overloading

► List of operators that can't be overloaded:

`.` `.*` `::` `?:` `#` `##`

► Reason: They take name, rather than value in their argument except for **?:**

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

Operator overloading

► The precedence of an operator is **NOT** affected due to overloading

► Example:

`c1*c2+c3`

`c3+c2*c1`

both yield the same answer

Operator overloading

► Associativity is **NOT** changed due to overloading

► Following arithmetic expression always is evaluated from left to right:

`c1 + c2 + c3 + c4`

Operator overloading

► 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`

Operator overloading

►Always write code representing the operator

►Example:

Adding subtraction code inside the + operator will create chaos

Operator overloading

►Creating a new operator is a syntax error (whether unary, binary or ternary)

► You cannot create $

Operator overloading

► Arity of an operator is NOT affected by overloading

► Example:

Division operator will take exactly two operands in any case:

b = c / d

Binary operators

► Binary operators act on two quantities

► Binary operators:

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

Binary operators

► General syntax:

Member function:

```
TYPE₁ CLASS::operator B_OP(
                         TYPE₂ rhs){
   ...
}
```

Binary operators

► General syntax:

Non-member function:

```
TYPE₁ operator B_OP(TYPE₂ lhs,
                         TYPE₃ rhs){
   ...
}
```

Binary operators

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

► Following is an error:

```
int operator + (int, int);
```

Binary operators

► Overloading + operator:

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

Binary operators

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

Binary operators

- ►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);
```

Binary operators

► If the return type was **void**,

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

Binary operators

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

Binary operators

► we have to do the same operation **`c1+c2+c3`** as:

`c1+c2`

`c1+c3`

// final result is stored in c1

Binary operators

► Drawback of void return type:

- Assignments and cascaded expressions are not possible
- Code is less readable
- Debugging is tough
- Code is very hard to maintain

Object-Oriented Programming (OOP)
Lecture No. 17

Binary operators

- Overloading + operator:

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

Binary operators

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

Binary operators

- 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);
```

Binary operators

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

► Example:

- In **`c1+c2`**, **`c1.operator+(c2)`**
- In **`c2+c1`**, **`c2.operator+(c1)`**

Binary operators

► The above examples don't handle the following situation:

```
Complex c1;
c1 + 2.325
```

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

Binary operators

► Modifying the complex class:

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

Binary operators

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

Binary operators

- ►Now suppose:

```
Complex c2, c3;
```

- ►We can do the following:

```
Complex c1 = c2 + c3;
```

and

```
Complex c4 = c2 + 235.01;
```

Binary operators

- ►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

Binary operators

►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);
}
```

Binary operators

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

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

Binary operators

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

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

Binary operators

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

Binary operators

► Other binary operators are overloaded very 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);
```

Assignment operator

► Consider a string class:

```
class String{
   int size;
   char * bufferPtr;
public:
   String();
   String(char *);
   String(const String &);
   …
};
```

Assignment operator

```
String::String(char * ptr){
    if(ptr != NULL){
        size = strlen(ptr);
        bufferPtr = new char[size+1];
        strcpy(bufferPtr, ptr);
    }
    else{
        bufferPtr = NULL; size = 0; }
}
```

Assignment operator

```
String::String(const String & rhs){
    size = rhs.size;
    if(rhs.size != 0){
        bufferPtr = new char[size+1];
        strcpy(bufferPtr, ptr);
    }
    else
        bufferPtr = NULL;
}
```

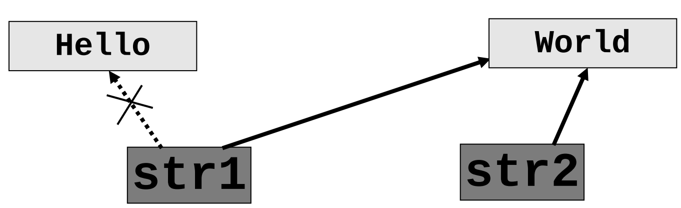
Assignment operator

```
int main(){
  String str1(“Hello");
  String str2(“World”);
  str1 = str2;
  return 0;
}
```

Member wise copy assignment

Assignment operator

► Result of **`str1 = str2`** (memory leak)



Assignment operator

- Modifying:

```
class String{
   …
public:
   …
   void operator =(const String &);
};
```

Assignment operator

```
void String::operator = (const String & rhs){
   size = rhs.size;
   if(rhs.size != 0){
      delete [] bufferPtr;
      bufferPtr = new char[rhs.size+1];
      strcpy(bufferPtr,rhs.bufferPtr);
   }
   else
         bufferPtr = NULL;
}
```

Assignment operator

```
int main(){
  String str1(“ABC");
  String str2(“DE”), str3(“FG”);
  str1 = str2;          // Valid…
  str1 = str2 = str3;  // Error…
  return 0;
}
```

Assignment operator

► **`str1=str2=str3`** is resolved as:

```
str1.operator=(str2.operator=
                    (str3))
```

Return type is void. Parameter can’t be void

Object-Oriented Programming (OOP)
Lecture No. 18

Assignment operator

► Modifying:

```
class String{
   …
public:
   …
   void operator =(const String &);
};
```

Assignment operator

```
void 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;
}
```

Assignment operator

```
int main(){
   String str1(“ABC");
   String str2(“DE”), str3(“FG”);
   str1 = str2;          // Valid…
   str1 = str2 = str3;  // Error…
   return 0;
}
```

Assignment operator

► **`str1=str2=str3`** is resolved as:

```
str1.operator=(str2.operator=
                      (str3))
```

Return type is void. Parameter can't be void

Assignment operator

► Solution: modify the **`operator =`** function as follows:

```
class String{
  …
public:
  …
  String & operator = (const
                    String &);
};
```

Assignment operator

```
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;
}
```

Assignment operator

```
void main(){
   String str1(“AB");
   String str2(“CD”), str3(“EF”);
   str1 = str2;
   str1 = str2 = str3;  // Now valid…
}
```

Assignment operator

- ► **`str1=str2=str3`** is resolved as:

```
str1.operator=(str2.operator=
                   (str3))
```

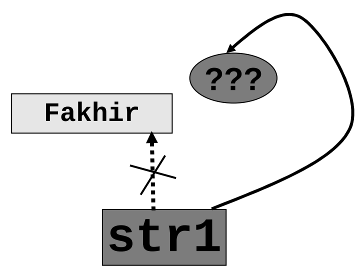
Return type is String .

Assignment operator

```
int main(){
  String str1("Fakhir");
  // Self Assignment problem…
  str1 = str1;
  return 0;
}
```

Assignment operator

►Result of **`str1 = str1`**




```
    …
  //   size = rhs.size;
  //   delete [] bufferPtr;
…
```

Assignment operator

```
String & String :: operator = (const
                                 String & rhs){
 if(this != &rhs){
   size = rhs.size;
   delete [] bufferPtr;
   if(rhs.bufferPtr != NULL){
      bufferPtr = new char[rhs.size+1];
      strcpy(bufferPtr,rhs.bufferPtr);
   }
   else bufferPtr = NULL;
 }
 return *this; }
```

Assignment operator

► Now self-assignment is properly handled:

```
int main(){
  String str1("Fakhir");
  str1 = str1;
  return 0;
}
```

Assignment operator

► Solution: modify the **operator=** function as follows:

```
class String{
  …
public:
  …
  const String & operator=
              (const String &);
};
```

Assignment operator

```
int main(){
   String s1("ABC"),
          s2("DEF"),
          s3("GHI");
   // Error…
   (s1 = s2) = s3;
   return 0;
}
```

Assignment operator

But we can do the following with primitive types:

```
int main(){
   int a, b, c;
   (a = b) = c;
   return 0;
}
```

Other Binary operators

► Overloading **+=** operator:

```
class Complex{
  double real, img;
public:
   Complex & operator+=(const Complex &
                              rhs);
   Complex & operator+=(count double &
                              rhs);
   ...
};
```

Other Binary operators

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

Other Binary operators

```
Complex & Complex::operator +=
           (const double & rhs){
  real = real + rhs;
  return * this;
}
```

Other Binary operators

```
int main(){
  Complex c1, c2, c3;
  c1 += c2;
  c3 += 0.087;
  return 0;
}
```

Operator overloading

►Friend functions minimize encapsulation

►This can result in:

- Data vulnerability
- Programming bugs
- Tough debugging

►Hence, use of friend functions must be limited

Operator overloading

►The + operator can be defined as a non-member, non-friend function:

```
Complex operator + (const Complex &
        a, const Complex & b){
   Complex t = a;
   return t += b
}
```

Operator overloading

```
Complex operator + (const double &
        a, const Complex & b){
   Complex t = b;
   return t += a;
}
```

Operator overloading

```
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

Object-Oriented Programming (OOP)
Lecture No. 19

Stream Insertion operator

► Often we need to display the data on the screen

► Example:

```
int i=1, j=2;
cout << “i= ”<< i << “\n”;
Cout << “j= ”<< j << “\n”;
```

Stream Insertion operator

```
Complex c1;
cout << c1;
cout << c1 << 2;
```

// Compiler error: binary '<<' : no operator // defined which takes a right-hand // operand of type 'class Complex'

Stream Insertion operator

```
class Complex{
    …
public:
    …
    void operator << (const
                    Complex & rhs);
};
```

Stream Insertion operator

```
int main(){
    Complex c1;
    cout << c1;      // Error
    c1 << cout;
    c1 << cout << 2; // Error
    return 0;
};
```

Stream Insertion operator

```
class Complex{
    …
public:
    …
    void operator << (ostream &);
};
```

Stream Insertion operator

```
void Complex::operator <<
        (ostream & os){

    os  << ‘(‘  << real

        << ‘,’ << img << ‘)’;

}
```

Stream Insertion operator

```
class Complex{

  ...

  friend ostream & operator <<
  (ostream & os, const Complex
                        & c);

};
```

Note: return type is NOT const

Note: this object is NOT const

Stream Insertion operator

// we want the output as: *(real, img)*

```
ostream & operator << (ostream &
          os, const Complex & c){
    os << ‘(‘ << c.real
       << ‘,‘
       << c.img << ‘)’;
    return os;
}
```

Stream Insertion operator

```
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 )
```

Stream Insertion operator

```
cout << c1 << c2;

is equivalent to

operator<<(
    operator<<(cout,c1),c2);
```

Stream Extraction Operator

► Overloading ">>" operator:

```
class Complex{
   ...
   friend istream & operator
   >> (istream & i, Complex &
        c);
};
```

Note: this object is NOT const

Stream Extraction Operator

```
istream & operator << (istream
        & in, Complex & c){
    in >> c.real;
    in >> c.img;
    return in;
}
```

Stream Extraction Operator

► 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;
```

Stream Extraction Operator

Output:

```
( 1.0025 , 0.0241 )
```

Other Binary operators

► Overloading comparison 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);
   …
};
```

Other Binary operators

```
bool Complex::operator ==(const
Complex & c){
   if((real == c.real) &&
      (img == c.img)){
        return true;
   }
   else
        return false;
}
```

Other Binary operators

```
bool operator ==(const
Complex& lhs, const Complex& rhs){
  if((lhs.real == rhs.real) &&
     (lhs.img == rhs.img)){
       return true;
  }
  else
       return false;
}
```

Other Binary operators

```
bool Complex::operator !=(const
Complex & c){
  if((real != c.real) ||
     (img != c.img)){
       return true;
  }
  else
       return false;
}
```

Object-Oriented Programming (OOP)
Lecture No. 20

Other Binary operators

- 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();
   ...
};
```

Other Binary Operators

```
int main(){
    String str1("Test");
    String str2;
    str2.SetString("Ping");
    return 0;
}
```

Other Binary Operators

► 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

Other Binary Operators

► Or, we can add a function which changes a character at nth location

```
class String{
  ...
public:
  void SetChar(char c, int pos);
  ...
};
```

Other Binary Operators

```
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**"*

Subscript Operator

► An elegant solution:

► Overloading the subscript "**[]**" operator

Subscript Operator

```
int main(){
    String str2;
    str2.SetString(“Ping”);
    str[2] = ‘o’;
    cout << str[2];
    return 0;
}
```

Subscript Operator

```
class String{
  ...
public:
  char & operator[](int);
  ...
};
```

Subscript Operator

```
char & String::operator[](
                          int pos){
  assert(pos>0 && pos<=size);
  return stringPtr[pos-1];
}
```

Subscript Operator

```
int main() {
  String s1(“Ping”);
  cout <<str.GetString()<< endl;
  s1[2] = ‘o’;
  cout << str.GetString();
  return 0;
}
```

Subscript Operator

- ►Output:

  `Ping`

  `Pong`

Overloading `()`

- ►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);
  ...
};
```

Function Operator

```
char & String::operator()
               (int pos){
  assert(pos>0 && pos<=size);
  return bufferPtr[pos-1];
}
```

Subscript Operator

```
int main(){
    String s1("Ping");
    char g = s1(2);       // g = 'i'
    s1(2) = 'o';
    cout << g << "\n";
    cout << str.GetString();
    return 0;
}
```

Function Operator

- Output:

```
i
Pong
```

Function Operator

```
class String{
    ...
public:
    String operator()(int, int);
    ...
};
```

Function Operator

```
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;
}
```

Function Operator

```
int main(){
    String s(“Hello World”);
    // “<<“ is overloaded
    cout << s(1, 5);
    return 0;
}
```

Function Operator

Output:

Hello

Unary Operators

► Unary operators:

- **& * + - ++ -- ! ~**

► 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;

Unary Operators

► General syntax for unary operators:

Member Functions:

TYPE & operator OP ();

Non-member Functions:

Friend TYPE & operator OP
(TYPE & t);

Unary Operators

► Overloading unary '-':

```
class Complex{
...
Complex  operator - ();
// friend Complex operator
//                    -(Complex &);
}
```

Unary Operators

► Member function definition:

```
Complex Complex::operator -(){
    Complex temp;
    temp.real = -real;
    temp.img = -img;
    return temp;
}
```

Unary Operators

```
Complex c1(1.0 , 2.0), c2;
c2 = -c1;
  // c2.real = -1.0
  // c2.img = -2.0
```

► Unary ‘+’ is overloaded in the same way

Object-Oriented Programming
(OOP)
Lecture No. 21

Unary Operators

- Unary operators are usually prefix, except for **++** and **--**
- **++** and **--** both act as prefix and postfix
- Example:
  - `h++;`
  - `g-- + ++h - --i;`

Unary Operators

- ►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 ++

Unary Operators

►Example:

```
int x = 1, y = 2;
cout << y++ << endl;
cout << y;
```

►Output:

2

3

Unary Operators

►Example:

```
int y = 2;
y++++;        // Error
y++ = x;      // Error
```

Post-increment ++ returns by value, hence an error while assignment

Unary Operators

►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 ++

Unary Operators

► Example:

```
int  y = 2;
cout << ++y << endl;
cout << y << endl;
```

► Output:

3
3

Unary Operators

► Example:

```
int x = 2, y = 2;
++++y;
cout << y;
++y = x;
cout << y;
```

Pre-increment ++ returns by reference, hence **NOT** an error

► Output:

4
2

Unary Operators

► Example (Pre-increment):

```
class Complex{
  double real, img;
public:
...
  Complex & operator ++ ();
// friend Complex & operator
//         ++(Complex &);
}
```

Unary Operators

► Member function definition:

```
Complex & Complex::operator++(){
  real = real + 1;
  return * this;
}
```

Unary Operators

► Friend function definition:

```
Complex & operator ++ (Complex
                        & h){
  h.real += 1;
  return h;
}
```

Unary Operators

```
Complex h1, h2, h3;
++h1;
```

► Function **`operator++()`** returns a reference so that the object can be used as an *lvalue*

```
++h1 = h2 + ++h3;
```

Unary Operators

► How does a compiler know whether it is a pre-increment or a post-increment ?

Unary Operators

► A post-fix unary operator is implemented using:

Member function with 1 dummy int argument

OR

Non-member function with two arguments

Unary Operators

► 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

Unary Operators

► Post-increment operator:

```
class Complex{
...
  Complex operator ++ (int);
// friend Complex operator
//   ++(const Complex &, int);
}
```

Unary Operators

- Member function definition:

```
Complex Complex::operator ++
                         (int){
  complex t = *this;
  real += 1;
  return t;
}
```

Unary Operators

- Friend function definition:

```
Complex operator ++ (const
           Complex & h, int){
  complex t = h;
  h.real += 1;
  return t;
}
```

Unary Operators

- ►The dummy parameter in the operator function tells compiler that it is post-increment
- ►Example:

```
Complex h1, h2, h3;
h1++;
h3++ = h2 + h3++; // Error…
```

Unary Operators

- ►The *pre* and *post* decrement operator `--` is implemented in exactly the same way

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:

C style type casting

```
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

Type Conversion

► Conversion from any other type to current type:
  - Requires a constructor with a single parameter

► Conversion from current type to any other type:
  - Requires an overloaded operator

Type Conversion

►Conversion from other type to current type (**int** to **String**):

```
class String{
  ...
public:
  String(int a);
  char * GetStringPtr()const;
};
```

Type Conversion

```
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;
}
```

Type Conversion

```
int main(){
   String s = 345;
   cout << s.GetStringPtr() << endl;
   return 0;
}
```

Type Conversion

- Output:

```
String(int) called…
345
```

Type Conversion

► Automatic conversion has drawbacks

► Conversion takes place transparently even if the user didn’t wanted the conversion

Type Conversion

User can write the following code to initialize the string with a single character:

```
int main(){
    String s = ‘A’;
    cout << s.GetStringPtr()<< endl
         << s.GetSize()   << endl;
    return 0;
}
```

Type Conversion

► Output:

`String(int) called…`

`65` ASCII code for '**A**' !!!

`2` String size is also 2 instead of 1

Type Conversion

There is a mechanism in C++ to restrict automatic conversions

► Keyword **`explicit`**

► Casting must be explicitly performed by the user

Type Conversion

► Keyword **`explicit`** only works with constructors

► Example:

```
class String{
  …
public:
  …
  explicit String(int);
};
```

Type Conversion

```
int main(){
    String s;
    // Error…
    s = ‘A’;
    return 0;
}
```

Type Conversion

```
int main(){
    String s1, s2;
    // valid, explicit casting…
    s1 = String(101);
    // OR
    s2 = (String)204;
    return 0;
}
```

Type Conversion

- There is another method for type conversion:

  "Operator overloading"

  (*Converting from current type to any other type*)

Type Conversion

► General Syntax:

$\texttt{TYPE}_1\texttt{::Operator TYPE}_2\texttt{();}$

► Must be a member function

► NO return type and arguments are specified

► Return type is implicitly taken to be **$TYPE_2$** by compiler

Type Conversion

► Overloading pre-defined types:

```
class String{
   …
public:
   …
   operator int();
   operator char *();
};
```

Type Conversion

```
String::operator int(){
    if(size > 0)
        return atoi(bufferPtr);
    else
        return -1;
}
```

Type Conversion

```
String::operator char *(){
    return bufferPtr;
}
```

Type Conversion

```
int main(){
    String s("2324");
    cout << (int)s << endl
         << (char *)s;
    return 0;
}
```

Type Conversion

Output:

```
2324
2324
```

Type Conversion

► User-defined types can be overloaded in exactly the same way

► Only prototype is shown below:

```
class String{
  …
  operator Complex();
  operator HugeInt();
  operator IntVector();
};
```

Type Conversion

► Modifying String class:

```
class String{
  …
public:
  …
  String(char *);
  operator int();
};
```

Type Conversion

```
int main(){
    String s(“Fakhir");
    // << is NOT overloaded
    cout << s;
    return 0;
}
```

Type Conversion

Output:

Junk Returned…

Type Conversion

► Modifying String class:

```
class String{
  …
public:
  …
  String(char *);
  int AsInt();
};
```

Type Conversion

```
int String::AsInt(){
    if(size > 0)
        return atoi(bufferPtr);
    else
        return -1;
}
```

Type Conversion

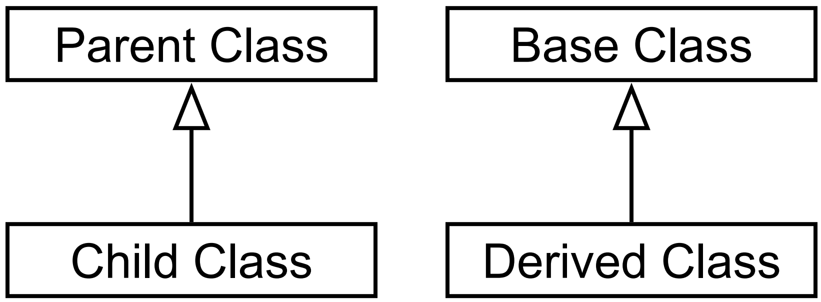
```
int main(){
    String s(“434");
    // << is NOT overloaded
    cout << s; //error
    cout << s.AsInt();
    return 0;
}
```

Object-Oriented Programming (OOP)
Lecture No. 22

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 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

UML Notation



Inheritance in C++

► There are three types of inheritance in C++

- Public
- Private
- Protected

“IS A” Relationship

- IS A relationship is modeled with the help of public inheritance
- Syntax

```
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

```
class Person{
  char *name;
  int age;
  ...
public:
  const char *GetName() const;
  int GetAge() const;
  ...
};
```

Example

```
class Student: public Person{
  int semester;
  int rollNo;
  ...
public:
  int GetSemester() const;
  int GetRollNo() const;
  void Print() const;
  ...
};
```

Example

```
void Student::Print()
{
  cout << name << " is in" << "
  semester " << semester;
}
```

ERROR

Example

```
void Student::Print()
{
  cout << GetName()
           << " is in semester "
     << semester;
}
```

Example

```
int main(){
  Student stdt;

  stdt.semester = 0;//error
  stdt.name = NULL; //error
  cout << stdt.GetSemester();
  cout << stdt.GetName();
  return 0;
}
```

Allocation in Memory

► The object of derived class is represented in memory as follows

Memory	
base member1 base member2 ...	Data members of base class
derived member1 derived member2 ...	Data members of derived class

Allocation in Memory

► 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

Constructors

base member1 base member2 ...	Base class constructor initializes the anonymous object
derived member1 derived member2 ...	Derived class constructor initializes the derived class object

Example

```
class Parent{
public:
  Parent(){ cout <<
     “Parent Constructor...”;}
};
class Child : public Parent{
public:
  Child(){   cout <<
     “Child Constructor...”;}
};
```

Example

```
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

Constructor

- 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
```

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
- 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
```