

Name of Institute: Indus Institute of Technology & Engineering

Name of Faculty: Ms. Hiral Shastri

Course code: CE0316

Course name: Object Oriented Programming with UML

Pre-requisites : Knowledge of C language will be useful.

Credit points: 4

Offered Semester: III

Course coordinator

Full name: Ms. Hiral Shastri

Department with sitting location: CSE dept, 4th floor Bhanwar Building.

Telephone:

Email: hiralshastri.cse@indusuni.ac.in

Consultation times: **Monday to Friday 3.10 pm to 5.00pm**

Course lecturer

Full name: Ms. Hiral shastri

Department with sitting location: CSE dept, 4th floor Bhanwar Building.

Email: hiralshastri.cse@indusuni.ac.in

Consultation times: **Monday to Friday 3.10 pm to 5.00pm**

Full name: Mr. Anand Trivedi

Department with sitting location: CSE dept, 4th floor Bhanwar Building.

Email: anandtrivedi.ce@indusuni.ac.in

Consultation times: **Monday to Friday 3.10 pm to 5.00pm**

Full name: Mr. Milan Bhadaliya

Department with sitting location: CSE dept, 4th floor Bhanwar Building.

Email: milanbhadaliya.ce@indusuni.ac.in

Consultation times: **Monday to Friday 3.10 pm to 5.00pm**

Students will be contacted throughout the session via mail with important information relating to this course.

Course Objectives

1. To learn the fundamental programming concepts and methodologies which are essential to building good C/C++ programs.
2. To write reusable modules, functions and classes as per Object Oriented Concepts.
3. To enhance employment of students, making good use of the object-oriented programming paradigm to simplify the design and implementation process
4. To encourage the practical problem solving skills.
5. To code, document, test, and implement a well-structured, robust computer program using the C/C++ programming language.

Course Outcomes (CO)

By participating and understanding all facets of this course a student will be able to:

1. Understand the difference between the top-down and bottom-up approach.
2. Describe the object-oriented programming approach in connection with C++.
3. Illustrate the process of data file manipulations using C++.
4. Apply the concepts of object-oriented programming.
5. Apply virtual and pure virtual function & complex programming situations.
6. Design and implement C++ programs for complex problems, making good use of the features of the language such as classes, inheritance and templates.

Course Outline

UNIT-I

[12 hours]

INTRODUCTION TO C++

Concepts of OOP: Introduction OOP, Procedural Vs. Object Oriented Programming, Principles of OOP, Benefits and applications of OOP C++Basics: Overview, Program structure, namespace, identifiers, variables, constants, enum, operators, typecasting, control structures C++ Functions: Simple functions, Call and Return by reference, Inline functions, Macro Vs. Inline functions, Overloading of functions, default arguments, friend functions.

UNIT-II

[12 hours]

Objects and classes

Basics of object and class in C++, Private, protected and public Members, static data and static function,

Constructors and their types, Destructors, Arrays & Strings: A standard C++ string class.

Operator Overloading: Overloading unary and binary operators, Operator Overloading with friend function, Data Conversion, type conversion, class to class, basic to class, class to basic

UNIT-III

[12 hours]

Concept of Inheritance

Types of inheritance: single, multiple, multilevel, hierarchical, hybrid, protected members, overriding, virtual base class, constructor in derived classes

Polymorphism: Pointers in C++, Pointers and Objects, this pointer, virtual and pure virtual functions, implementing polymorphism

I/O management: Concept of streams, cin and cout objects, C++ stream classes, Unformatted and formatted I/O, manipulators

UNIT-IV

[12 hours]

File management:

File stream, C++ File stream classes, File management functions, File modes, Binary and random files

Object-oriented Design

Object modeling using UML, Three models, Class Model (Object and Class Diagram), State model (state Diagram) and Interaction model (Use case diagrams, Activity diagrams, Interaction diagrams).

Method of delivery

Chalk and Board, PowerPoint presentation

Study time

3 Hours theory, 2 Hours practical

CO-PO Mapping (PO: Program Outcomes)

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	2	-	-	-	-	-	-	-	-	-	-	-
CO2	3	1	1	-	-	-	-	-	-	-	-	-
CO3	3	3	2	-	-	-	-	-	-	-	-	-
CO4	2	1	-	-	-	-	-	-	-	-	-	-
CO5	3	3	2	-	-	-	-	-	-	-	-	-
CO6	3	3	2	-	-	-	-	-	-	-	-	-

Blooms Taxonomy and Knowledge retention (For reference)

(Blooms taxonomy has been given for reference)

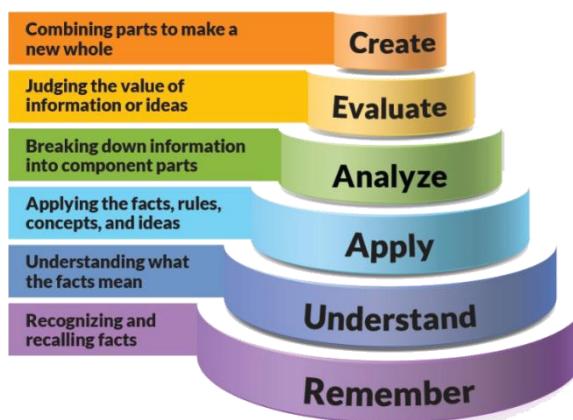


Figure 1: Blooms Taxonomy

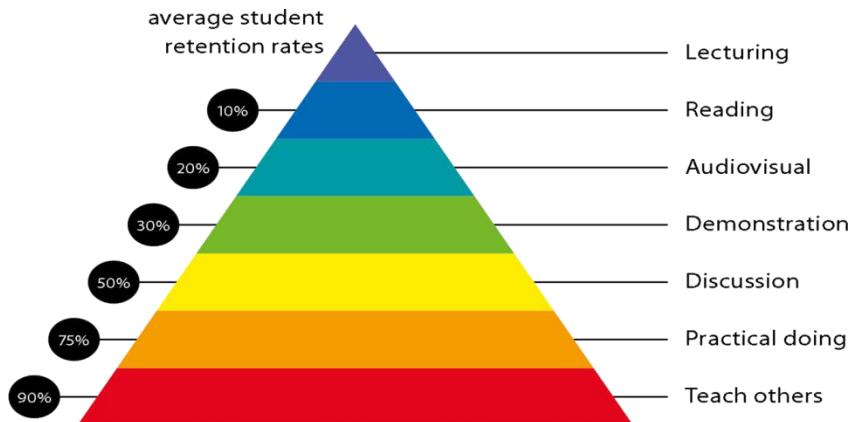


Figure 2: Knowledge retention

Graduate Qualities and Capabilities covered

(Qualities graduates harness crediting this Course)

General Graduate Qualities	Specific Department of _____ Graduate Capabilities
Informed Have a sound knowledge of an area of study or profession and understand its current issues, locally and internationally. Know how to apply this knowledge. Understand how an area of study has developed and how it relates to other areas.	1 Professional knowledge, grounding & awareness
Independent learners Engage with new ideas and ways of thinking and critically analyze issues. Seek to extend knowledge through ongoing research, enquiry and reflection. Find and evaluate information, using a variety of sources and technologies. Acknowledge the work and ideas of others.	2 Information literacy, gathering & processing
Problem solvers Take on challenges and opportunities. Apply creative, logical and critical thinking skills to respond effectively. Make and implement decisions. Be flexible, thorough, innovative and aim for high standards.	4 Problem solving skills
Effective communicators Articulate ideas and convey them effectively	5 Written communication 6 Oral communication

using a range of media. Work collaboratively and engage with people in different settings. Recognize how culture can shape communication.	7 Teamwork
<p>Responsible</p> <p>Understand how decisions can affect others and make ethically informed choices. Appreciate and respect diversity. Act with integrity as part of local, national, global and professional communities.</p>	10 Sustainability, societal & environmental impact

Practical work:

1	<p>Basics of programming</p> <p>1.1 write a program to print student name,address,college name,roll no.</p> <p>1.2 Write a program to perform addition,subtraction,multiplication and division of two number.</p> <p>1.3 Write a program to perform typecasting.</p>	To understand how C++ improves C with object-oriented features.
2	<p>2.1 Write a program to calculate the area of circle, rectangle and square using function overloading.</p> <p>2.2 Write a program to demonstrate the use of default arguments in function overloading.</p> <p>2.3 Write a program to demonstrate the use of returning a reference variable.</p>	To learn how to overload functions and operators in C++.
3	<p>3.1 Create a class student which stores the detail about roll no, name, marks of 5 subjects, i.e. science, Mathematics, English, C++. The class must have the following:</p> <ul style="list-style-type: none"> • Get function to accept value of the data members. • Display function to display values of data members. • Total function to add marks of all 5 subjects and store it in the data members named total. 	To learn how to design C++ classes for code reuse.

	<p>3.2 Create a function power() to raise a number m to power n. the function takes a double value for m and int value for n, and returns the result correctly. Use the default value of 2 for n to make the function calculate squares when this argument is omitted. Write a main that gets the values of m and n from the user to test the function.</p> <p>3.3 Write a basic program which shows the use of scope resolution operator.</p> <p>3.4 Write a C++ program to swap the value of private data members from 2 different classes.</p>	
4	<p>4.1 Write a program to illustrate the use of this pointer.</p> <p>4.2 An election is contested by five candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and count the votes cast for each candidate using an array variable count. In case a number is read outside the range of 1 to 5, the ballot should be considered as a ‘spoilt ballot’ and the program should also count the number of spoilt ballots.</p> <p>4.3 Write a program to call member functions of class in the main function using pointer to object and pointer to member function.</p>	To learn how to design C++ pointers
5	<p>5.1 Using friend function find the maximum number from given two numbers from two different classes. Write all necessary functions and constructors for the program.</p> <p>5.2 Using a friend function, find the average of three numbers from three different classes. Write all necessary member functions and constructor for the classes.</p> <p>5.3 Define currency class which contains rupees and paisa as data members. Write a friend function named AddCurrency () which add 2 different Currency objects and returns a Currency object. Write parameterized constructor to initialize the values and use appropriate functions to get the details from the user and display it.</p> <p>5.4 Create Calendar class with day, month and year as data members. Include default and parameterized constructors to initialize a Calendar object with a valid date value. Define a function AddDays to add days to the Calendar object. Define a display function to show data in “dd/mm/yyyy” format.</p>	To learn how to implement constructors and class member functions.

6	<p>6.1 Create a class named ‘String’ with one data member of type char *, which stores a string. Include default, parameterized and copy constructor to initialize the data member. Write a program to test this class.</p> <p>6.2 Write a base class named Employee and derive classes Male employee and Female Employee from it. Every employee has an id, name and a scale of salary. Make a function ComputePay (in hours) to compute the weekly payment of every employee. A male employee is paid on the number of days and hours he works. The female employee gets paid the wages for 40 hours a week, no matter what the actual hours are. Test this program to calculate the pay of employee.</p> <p>6.3 Create a class called scheme with scheme_id, scheme_name, outgoing_rate, and message charge. Derive customer class form scheme and include cust_id, name and mobile_no data. Define necessary functions to read and display data. Create a menu driven program to read call and message information for a customer and display the detail bill.</p>	To learn how to implement copy constructors and class member functions.
7	<p>7.1 Write a program with use of inheritance: Define a class publisher that stores the name of the title. Derive two classes book and tape, which inherit publisher. Book class contains member data called page no and tape class contain time for playing. Define functions in the appropriate classes to get and print the details.</p> <p>7.2 Create a class account that stores customer name, account no, types of account. From this derive classes cur_acc and sav_acc to include necessary member function to do the following:</p> <ul style="list-style-type: none"> • Accepts deposit from customer and update balance • Compute and Deposit interest • Permit withdrawal and Update balance. <p>7.3 Write a base class named Employee and derive classes Male employee and Female Employee from it. Every employee has an id, name and a scale of salary. Make a function ComputePay (in hours) to compute the weekly payment of every employee. A male employee is paid on the number of days and hours he works. The female employee gets paid the wages for 40 hours a week, no matter what the actual hours are. Test this program to calculate the pay of employee.</p>	To learn how containment and inheritance promote code reuse in C++.
8	<p>8.1 Create a class vehicle which stores the vehicleno and chassisno as a member. Define another class for scooter, which inherits the data members of the class vehicle and has a data member for a storing wheels and company. Define another class for which inherits the data member of the</p>	To learn how inheritance and virtual functions

	<p>class vehicle and has a data member for storing price and company. Display the data from derived class. Use virtual function.</p> <p>8.2 Create a base class shape. Use this class to store two double type values that could be used to compute the area of figures. Derive two specific classes called triangle and rectangle from the base shape. Add to the base class, a member function <code>get_data()</code> to initialize the base class data members and another member function <code>display_area()</code> to compute and display the area of figures. Make <code>display_area()</code> as a virtual function and redefine this function in the derived class to suit their requirements.</p> <p>8.3 Write a program to demonstrate the use of pure virtual function.</p> <p>8.4 For multiple inheritance, write a program to show the invocation of constructor and destructor.</p> <p>8.5 Create a class string with character array as a data member and write a program to add two strings with use of operator overloading concept.</p> <p>8.6 Create a class distance which contains feet and inch as a data member. Overhead <code>==</code>, <code><and></code> operator for the same class. Create necessary functions and constructors too.</p>	<p>implement dynamic binding with polymorphism.</p>
9	<p>9.1 Create a class MATRIX of size mxn. Overload <code>+</code> and <code>-</code> operators for addition and subtraction of the MATRIX.</p> <p>9.2 Define a class Coord, which has <code>x</code> and <code>y</code> coordinates as its data members. Overload <code>++</code> and <code>-</code> operators for the Coord class. Create both its prefix and postfix forms.</p> <p>9.3 Create one class called Rupees, which has one member data to store amount in rupee and create another class called Paise which has member data to store amount in paise. Write a program to convert one amount to another amount with use of type conversion.</p> <p>9.4 Create two classes Celsius and Fahrenheit to store temperature in terms of Celsius and Fahrenheit respectively. Include necessary functions to read and display the values. Define conversion mechanism to convert Celsius object to Fahrenheit object and vice versa. Show both types of conversions in main function.</p>	<p>To learn how to overload functions and operators in C++.</p>
10	<p>10.1 Write a program to create a function template for finding maximum value contained in an array.</p>	<p>To learn how to design and implement</p>

	<p>10.2 Write a program to create a class template for the ‘Array’ class.</p> <p>10.3 Create a template for the bubble sort function.</p> <p>10.4 Write a program to illustrate the use of insertion and extraction operators for Text mode Input/Output.</p>	generic classes with C++ templates.
11	<p>11.1 Write a program to illustrate the use of put(), get() and getline() functions for Text mode Input/Output.</p> <p>11.2 Write a program to illustrate the use of read() and write() functions for Binary mode Input/Output.</p> <p>11.3 Write a program to illustrate the use of manipulators in file handling.</p> <p>11.4 Write down a program to Copy source file ‘source.txt’ to destination file.</p> <p>11.5 A file contains a list of telephone numbers in the following format:</p> <p>a) Ram 47890</p> <p>b) Krishna 878787</p> <p>c) -----</p> <p>d) -----</p> <p>The names contain only one word and the names and telephone numbers are separated by white space. Write a Program to read the tel.dat file and display the content. The names should be left justified and the number right-justified.</p>	To learn how to design and implement files with C++.

Attendance Requirements

The University norms states that it is the responsibility of students to attend all lectures, tutorials, seminars and practical work as stipulated in the course outline. Minimum attendance requirement as per university norms is compulsory for being eligible for semester examinations.

Text books

1. Object oriented Programming with C++ ,Balaguruswamy, Tata Mcgraw Hill Publication Co. Ltd 2000.
2. Object oriented programming in turbo C++ ,RobbetLofre, Galgotia Publication Pvt Ltd. 1994.

Reference Books:

1. The Complete Reference C++ , Fourth Edition , Herbert Schildt , Tata Mcgraw Hill Publication.
2. The C++ programming language , BjarneStroustrup ,Addison

Additional Materials

Web Resource

<https://nptel.ac.in/courses/106105082/>

<https://nptel.ac.in/downloads/106105080/>

ASSESSMENT GUIDELINES

Your final course mark will be calculated from the following:

Theory [Total -100]	Practical [Total -100]
CIE Total :60 Mid Semester Exam: 40 Marks Attendance: 05 Marks Quiz : 10 Marks Assignment:05 Marks	CIE Total 60 Internal Practical Exam : 20 marks Viva :10 Marks Regularity+Attendance + Performance : 15 marks Practical file :15 Marks
ESE total: 40 Marks	ESE total : 40 Marks

SUPPLEMENTARY ASSESSMENT

Students who receive an overall mark less than 40% in internal component or less than 40% in the end semester will be considered for supplementary assessment in the respective components (i.e internal component or end semester) of semester concerned. Students must make themselves available during the supplementary examination period to take up the respective components (internal component or end semester) and need to obtain the required minimum 40% marks to clear the concerned components.

Practical Work Report/Laboratory Report:

A report on the practical work is due the subsequent week after completion of the class by each group.

Late Work

Late assignments will not be accepted without supporting documentation. Late submission of the reports will result in a deduction of -% of the maximum mark per calendar day

Format

All assignments must be presented in a neat, legible format with all information sources correctly referenced. **Assignment material handed in throughout the session that is not neat and legible will not be marked and will be returned to the student.**

Retention of Written Work

Written assessment work will be retained by the Course coordinator/lecturer for two weeks after marking to be collected by the students.

University and Faculty Policies

Students should make themselves aware of the University and/or Faculty Policies regarding plagiarism, special consideration, supplementary examinations and other educational issues and student matters.

Plagiarism - Plagiarism is not acceptable and may result in the imposition of severe penalties. Plagiarism is the use of another person's work, or idea, as if it is his or her own - if you have any doubts at all on what constitutes plagiarism, please consult your Course coordinator or lecturer. Plagiarism will be penalized severely.

Do not copy the work of other students.

Do not share your work with other students (except where required for a group activity or assessment.

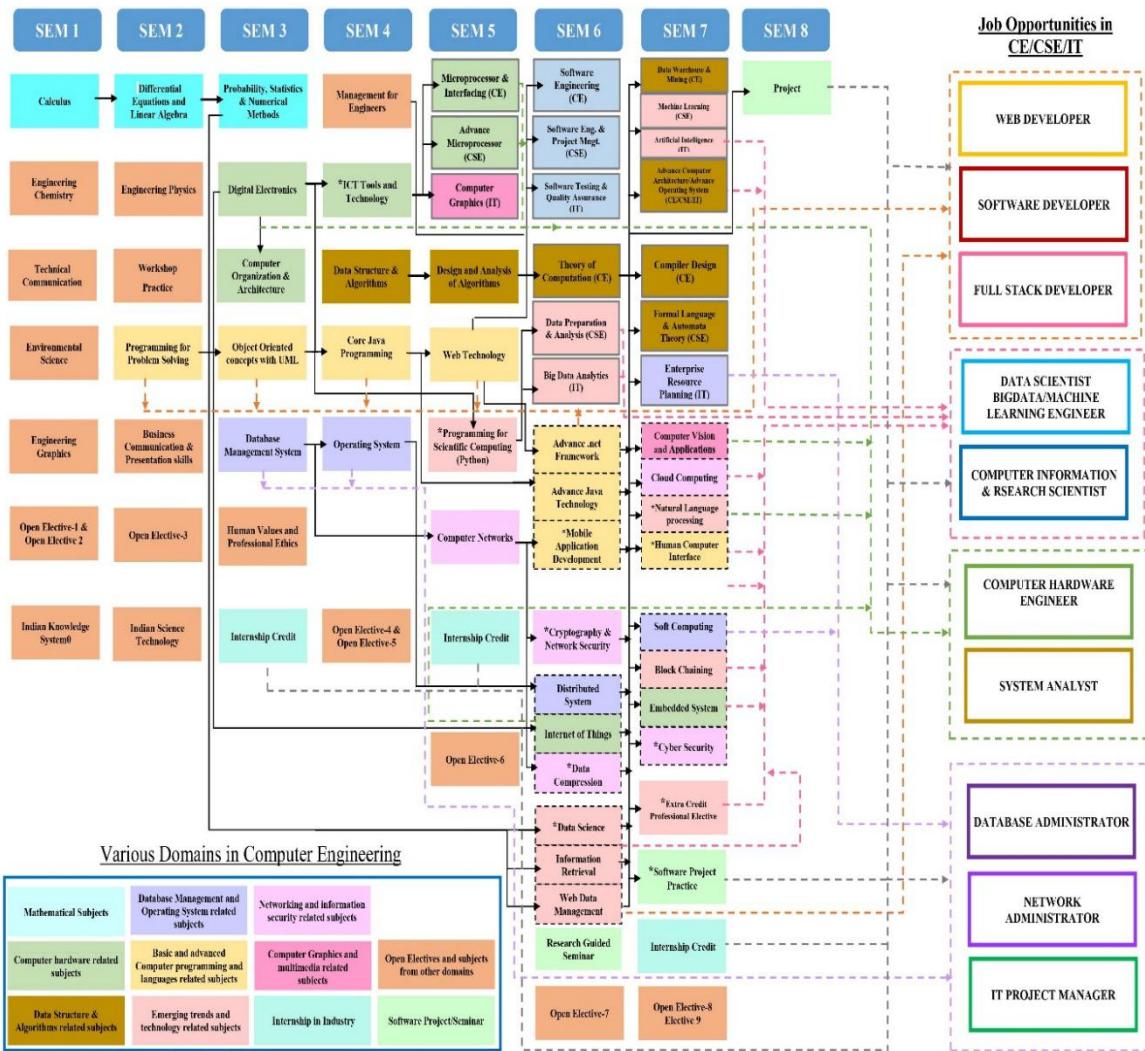
Course schedule (subject to change)

(Mention quiz, assignment submission, breaks etc as well in the table under the Teaching Learning Activity Column)

	Week #	Topic & contents	CO Addressed	Teaching Learning Activity (TLA)
	Week 1	Concepts of OOP: Introduction OOP, Procedural Vs. Object Oriented Programming, Principles of OOP, Benefits and applications of OOP	I	Chalk & Board, Discussion
	Week 2	C++Basics: Overview, Program structure, namespace, identifiers, variables, constants, enum, operators, typecasting, control structures C++ Functions: Simple functions, Call and Return by reference, Inline functions, Macro Vs. Inline functions	I	Presentation, Chalk & Board
	Week 3	Overloading of functions, default arguments, friend functions	I	Presentation, Chalk & Board
	Week 4	Objects and classes: Basics of object and class in C++, Private, protected and public Members,	II	Presentation, Chalk & Board
	Week 5	static data and static function, Constructors and their types , Destructors, Arrays & Strings: A standard C++ string class.	II	Presentation, Chalk & Board
	Week 6	Operator Overloading: Overloading unary and binary operators, Operator Overloading with friend function,	II	<i>Model presentation</i>
	Week 7	Data Conversion, type conversion, class to class, basic to class, class to basic	II	Presentation, Chalk & Board, Demonstration
	Week 8	Concept of Inheritance, types of inheritance: single, multiple, multilevel, hierarchical, hybrid, protected	II	Presentation, Chalk & Board, Demonstration

		members,		
Week 9		overriding, virtual base class, constructor in derived classes	<i>III,V</i>	Presentation, Chalk & Board
Week 10		Polymorphism: Pointers in C++, Pointers and Objects, this pointer, virtual and pure virtual functions, implementing polymorphism	<i>III,V</i>	Presentation, Chalk & Board
Week 11		I/O management: Concept of streams, cin and cout objects, C++ stream classes, Unformatted and formatted I/O, manipulators	<i>IV,VI</i>	Presentation, Chalk & Board
Week 12		File management: File stream, C++ File stream classes, File management functions, File modes, Binary and random files	<i>III,V</i>	Presentation, Chalk & Board
Week 13		Object-oriented Design Object modeling using UML, Three models	<i>IV</i>	Presentation, Chalk & Board
Week 14		Class Model (Object and Class Diagram), State model (state Diagram) and Interaction model (Use case diagrams, Activity diagrams, Interaction diagrams).	<i>IV</i>	Presentation, Chalk & Board
Week 15		Revision	<i>IV</i>	Presentation, Chalk & Board

COMPUTER ENGINEERING DEPARTMENT COURSE DEPENDANCY CHART



Unit-1

Principal of OOP

Concepts of OOP

- Introduction to OOP
- Procedural Vs. Object Oriented Programming
- Principles of OOP
- Benefits and applications of OOP

Introduction to OOP

- *OOP* is a design philosophy. It stands for Object Oriented Programming.
- C++ was founded in (1983)



Bjarne Stroustrup

Introduction to OOP

- Object-Oriented Programming (*OOP*) uses a different set of programming languages than old procedural programming languages like (*C, Pascal, etc.*).
- Everything in *OOP* is grouped as self sustainable "*objects*".

What is Object?



Pen



Board



Laptop



Bench



Student



Projector

Physical objects...

What is Object?

SEARCH RESULT:

Name : SHAH BHAUTIK VIRENBHAI
Enrollment No. : 110280106055
Exam Seat No. : E814875
Exam : BE SEM 8 - Regular (MAY 2015)
Branch : CIVL ENGINEERING

Declared On : 19 Jun 2015

SUBJECT CODE	SUBJECT NAME	GRADE	INT. GRADE	ABSENT	BACKLOG
180601	Design Of Hydraulic Structures	BC	N	N	N - N - N - N
180602	Dock Harbour & Airport Engineering	BB	N	N	N - N - N - N
180603	Professional Practice & Valuation	BB	N	N	N - N - N - N
180604	Structural Design-II	BC	N	N	N - N - N - N
180605	Project -II	AA	N	N	N - N - N - N
180607	Repairs & Rehabilitation Of Structures	BB	N	N	N - N - N - N

Current Sem Backlog: 0 Total Backlog: 0 SPI: 8.20 CPI: 7.58 CGPA: 7.98

Backlog : Sem-1: 0 | Sem-2: 0 | Sem-3: 0 | Sem-4: 0 | Sem-5: 0 | Sem-6: 0 | Sem-7: 0 | Sem-8: 0 |

Online Re-Check/Re-Assessment: from 19-06-2015 to 24-06-2015 Students Guid
please send recheck query to respected department [BE,BPharm,PDDC,HM - be@gtu.edu.in] [Diploma,
DipPharm - dippharm@gtu.edu.in] [ME,MPH,MBA,MCA - pg@gtu.edu.in]. Rules of Reassessment

Note : This is a computer generated mark-sheet. Printed On : Friday, June 19, 2015 - 2:53:26 PM

Congratulations!! You have passed this exam.

Result



Bank Account

Logical objects...



Account

Attributes and operations

**Attributes:**

Name
Age
Weight

Attributes:

Company
Model
Weight

Attributes:

AccountNo
HolderName
Balance

Operations:

Eat
Sleep
Walk

Operations:

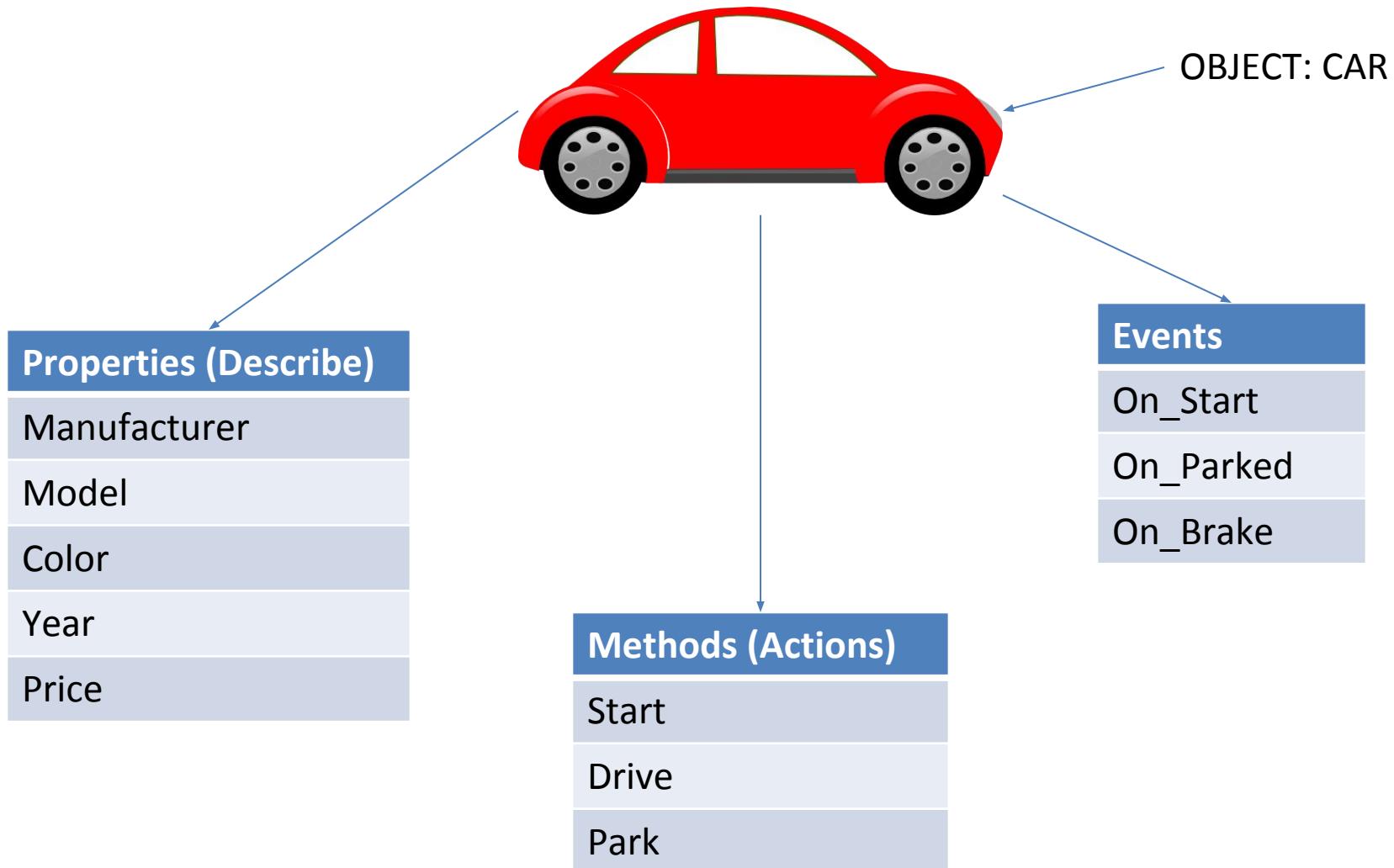
Drive
Stop
FillFuel

Operations:

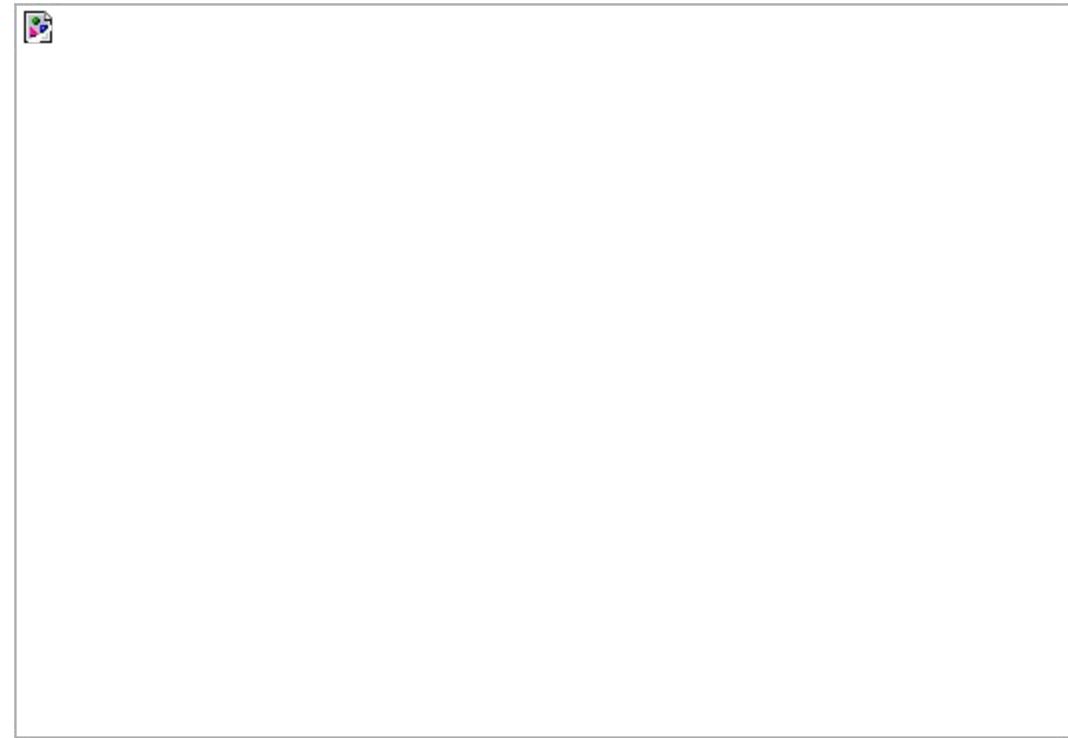
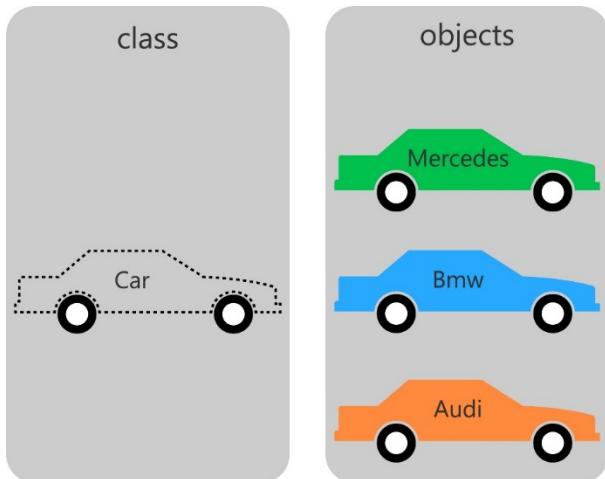
Deposit
Withdraw
Transfer

Write down 5 objects with its attributes and operations

What is Object ?

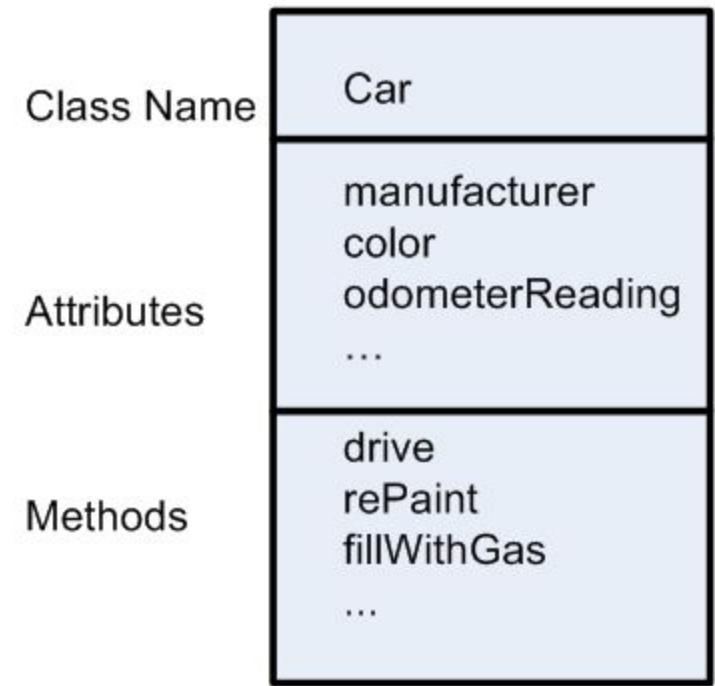
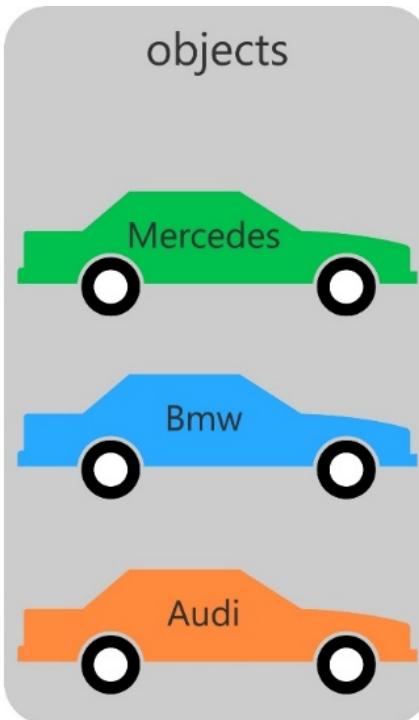
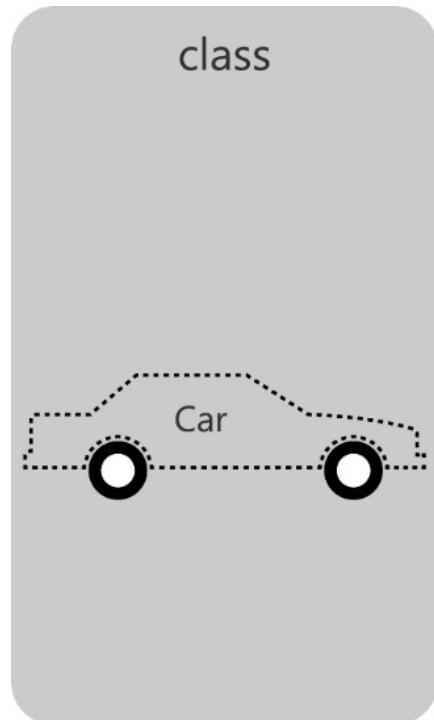


Classes...



Class: Blueprint (template) for object.
Object: Instance of class.

Class



Applications of OOP

- Real Time Systems Design
- Simulation and Modeling System
- Object Oriented Database
- Client-Server System
- Neural Networking and Parallel Programming
- Decision Support and Office Automation Systems
- CIM/CAD/CAM Systems
- AI and Expert Systems

Procedural Vs. Object Oriented Programming

POP	OOP
Emphasis is on doing things not on data, means it is function driven	Emphasis is on data rather than procedure, means object driven
Main focus is on the function and procedures that operate on data	Main focus is on the data that is being operated
Top Down approach in program design	Bottom Up approach in program design
Large programs are divided into smaller programs known as functions	Large programs are divided into classes and objects
Most of the functions share global data	Data is tied together with function in the data structure

Procedural Vs. Object Oriented Programming

POP	OOP
Data moves openly in the system from one function to another function	Data is hidden and cannot be accessed by external functions
Adding of data and function is difficult	Adding of data and function is easy
We cannot declare namespace directly	We can use name space directly, Ex: using namespace std;
Concepts like inheritance, polymorphism, data encapsulation, abstraction, access specifiers are not available.	Concepts like inheritance, polymorphism, data encapsulation, abstraction, access specifiers are available and can be used easily
Examples: C, Fortran, Pascal, etc...	Examples: C++, Java, C#, etc...

Principles of OOP (A.E.I.P)

- There are mainly four OOP Principles

Abstraction

Encapsulation

Inheritance

Polymorphism

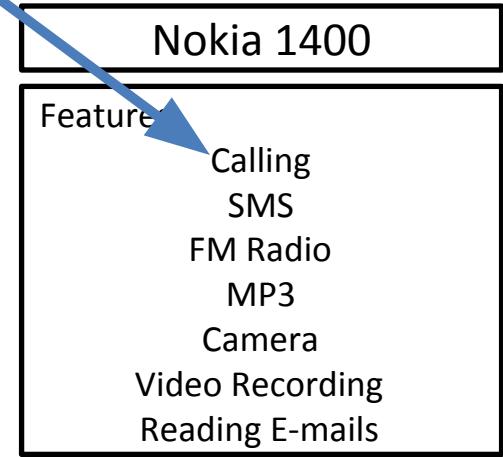
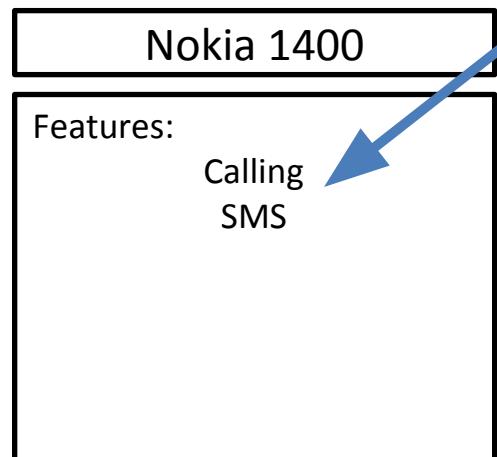
Abstraction

- **Abstraction** refers to the act of representing essential features without including the background details or explanations.
- **Abstraction** provides you a generalized view of your classes or object by providing relevant information.
- **Abstraction** is the process of hiding the working style of an object, and showing the information of an object in understandable manner.

Abstraction Example



Abstract information (Necessary and Common Information) for the object “Mobile Phone” is make a call to any number and can send SMS.”



Abstraction Example

- Example:
If somebody in your collage tell you to fill application form, you will fill your details like **name, address, data of birth, which semester, percentage you have got** etc.
- If some doctor gives you an application to fill the details, you will fill the details like **name, address, date of birth, blood group, height and weight**.
- See in the above example what is the common thing?
Age, name, address so you can create the class which consist of common thing that is called abstract class.
That class is not complete and it can inherit by other class.

Encapsulation

- The wrapping up of data and functions into a single unit is known as **encapsulation**
- The insulation of the data from direct access by the program is called **data hiding** or **information hiding**.
- It is the process of enclosing one or more details from outside world through access right.

Encapsulation



- **Encapsulation** is the process of combining data and functions into a single unit called class. In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class.
- Users are unaware about working of circuitry and hardware devices.

- **Abstraction** is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.
- Consider your mobile phone, you just need to know what buttons are to be pressed to send a message or make a call, What happens when you press a button, how your messages are sent, how your calls are connected is all abstracted away from the user.

Abstraction Vs Encapsulation

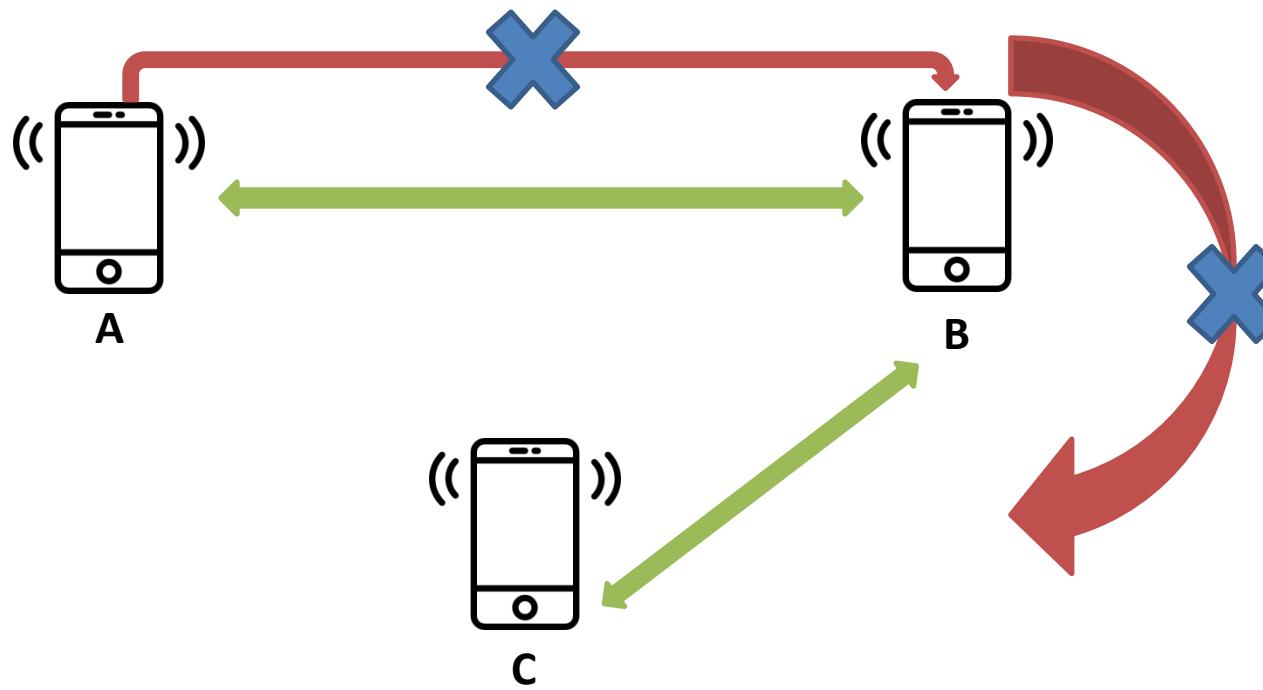
- Abstraction says what details to be made visible & Encapsulation provides the level of access right to that visible details.

Example:

- When we switch on the Bluetooth I am able to connect another mobile but not able to access the other mobile features like dialling a number, accessing inbox etc. This is because, Bluetooth feature is given some level of **abstraction**.

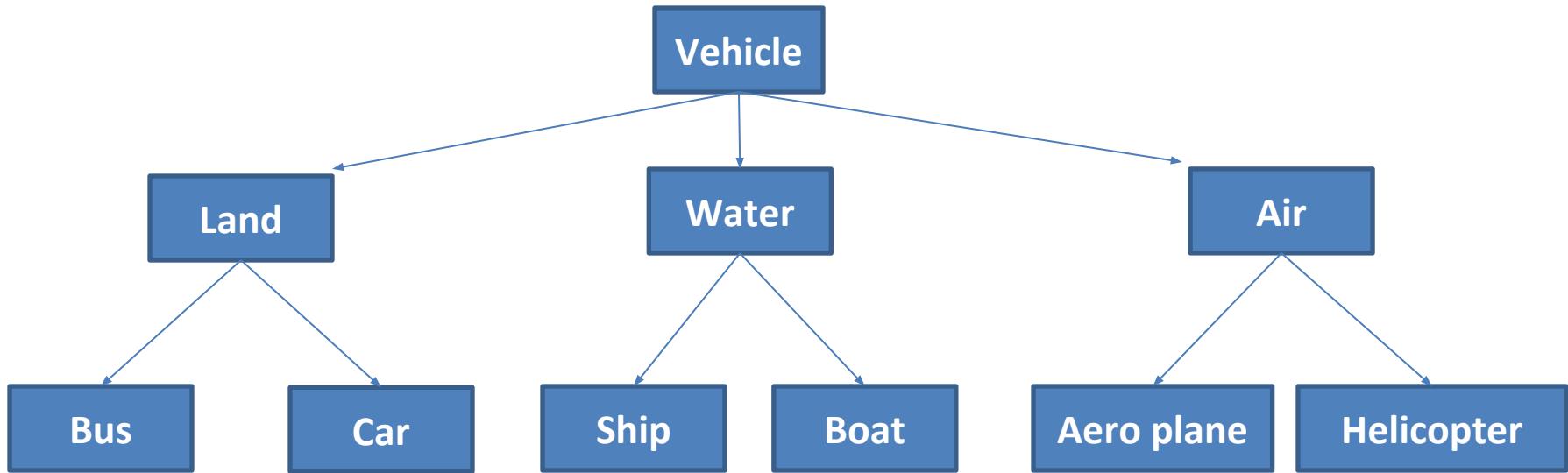
Abstraction Vs Encapsulation

- When mobile A is connected with mobile B via Bluetooth whereas mobile B is already connected to mobile C then A is not allowed to connect C via B. This is because of accessibility restriction.



Inheritance

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



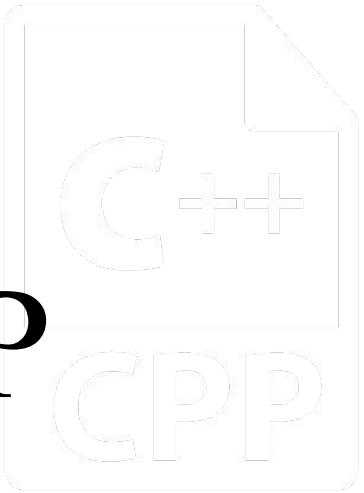
- Here Vehicle class can have properties like Chassis no. , Engine, Colour etc.
- All these properties inherited by sub classes of vehicle class.

Polymorphism

- **Polymorphism** means ability to take more than one form.
- For example the operation **addition**.
- For two numbers the operation will generate a **sum**.
- If the operands are strings, then the operation would produce a third string by **concatenation**.

Unit-1

Principal of OOP



Simple C++ Program

A Simple C++ Program

```
#include <iostream> //include header file
using namespace std;
int main()
{
    cout << "Hello World"; // C++ statement
    return 0;
}
```

- **iostream** is just like we include **stdio.h** in c program.
- It contains declarations for the identifier **cout** and the insertion operator **<<**.
- **iostream** should be included at the beginning of all programs that use input/output statements.

A Simple C++ Program (Cont...)

```
#include <iostream> //include header file
using namespace std;
int main()
{
    cout << "Hello World"; // C++ statement
    return 0;
}
```

- A namespace is a declarative region.
- A **namespace** is a part of the program in which certain names are recognized; outside of the namespace they're unknown.
- namespace defines a scope for the identifiers that are used in a program.
- **using** and **namespace** are the keywords of C++.

A Simple C++ Program (Cont...)

```
#include <iostream> //include header file
using namespace std;
int main()
{
    cout << "Hello World"; // C++ statement
    return 0;
}
```

- **std** is the namespace where ANSI C++ standard class libraries are defined.
- Various program components such as **cout**, **cin**, **endl** are defined within **std** namespace.
- If we don't use the **using** directive at top, we have to add the **std** followed by **::** in the program before identifier.

```
std::cout << "Hello World";
```

A Simple C++ Program (Cont...)

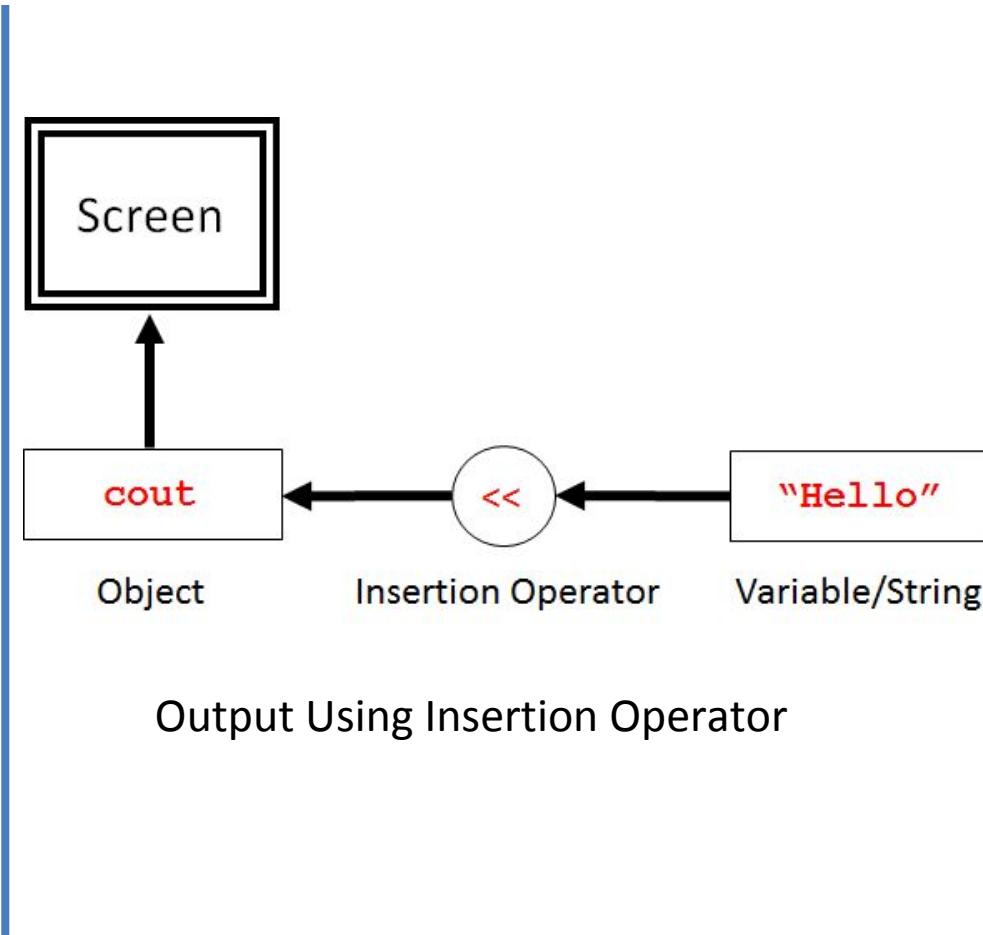
```
#include <iostream> //include header file
using namespace std;
int main()
{
    cout << "Hello World"; // C++ statement
    return 0;
}
```

- In C++, **main()** returns an integer type value.
- Therefore, every **main()** in C++ should end with a **return 0;** statement; otherwise error will occur.
- The return value from the **main()** function is used by the runtime library as the **exit code** for the process.

Insertion Operator <<

```
cout << "Hello World";
```

- The operator `<<` is called the insertion operator.
- It inserts the contents of the variable on **its right** to the object on **its left**.
- The identifier `cout` is a predefined object that represents standard output stream in C++.
- Here, Screen represents the output. We can also redirect the output to other output devices.
- The operator `<<` is used as bitwise left shift operator also.



Program: Basic C++ program

Write a C++ Program to print following

Name: demo

City: ahmedabad

Country: India

Program: Basic C++ program

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Name: demo";
    cout << "City: ahmedabad";
    cout << "Country: India";
    return 0;
}
```

Output

Name: demo City: ahmedabad Country: India

Program: Basic C++ program(Cont...)

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Name: demo\n";
    cout << "City: ahmedabad\n";
    cout << "Country: India";
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Name: demo" << endl;
    cout << "City: ahmedabad" << endl;
    cout << "Country: India" << endl;
    return 0;
}
```

Output

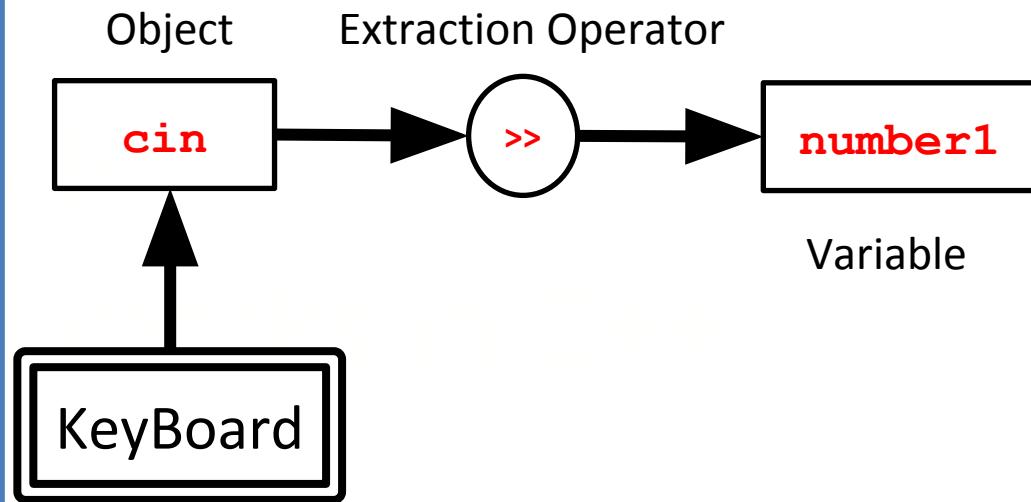
Name: demo
City: ahmedabad
Country: India

- The **endl** manipulator and **\n** has same effect. Both inserts new line to output.
- But, difference is **endl** immediate flush to the output while **\n** do not.

Extraction Operator >>

```
cin >> number1;
```

- The operator **>>** is called the extraction operator.
- It extracts (or takes) the value **from keyboard** and assigns it to the variable on **its right**.
- The identifier **cin** is a predefined object that represents standard input stream in C++.
- Here, standard input stream represents the Keyboard.
- The operator **>>** is used as bitwise right shift operator also.



Program: Basic C++ program

```
#include<iostream>
using namespace std;
int main()
{
    int number1,number2;

    cout<<"Enter First Number: ";
    cin>>number1;                      //accept first number

    cout<<"Enter Second Number: ";
    cin>>number2;                      //accept first number

    cout<<"Addition : ";
    cout<<number1+number2;              //Display Addition
    return 0;
}
```

C++ Tokens

C++ Tokens

- The smallest individual unit of a program is known as **token**.
- C++ has the following tokens:
 - Keywords
 - Identifiers
 - Constants
 - Strings
 - Special Symbols
 - Operators

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World";
    return 0;
}
```

Keywords and Identifier

- C++ reserves a set of 84 words for its own use.
- These words are called **keywords** (or reserved words), and each of these keywords has a special meaning within the C++ language.
- **Identifiers** are names that are given to various user defined program elements, such as variable, function and arrays.
- Some of Predefined **identifiers** are cout, cin, main

We cannot use Keyword as user defined identifier.

Keywords in C++

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typeof
class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual
default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while

Rules for naming identifiers in C++

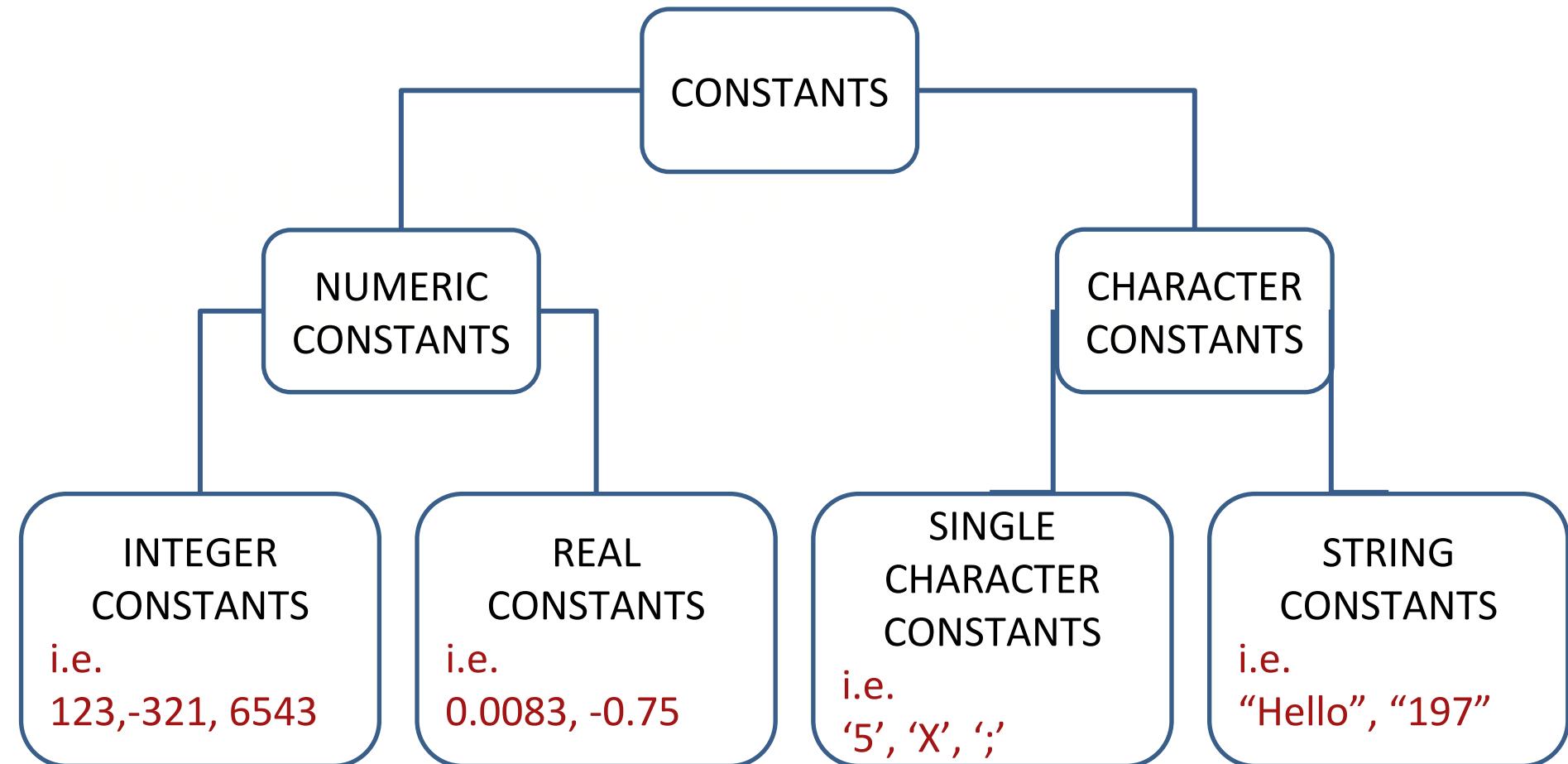
1. First Character must be an **alphabet or underscore**.
2. It can contain **only letters**(a..z A..Z), **digits**(0 to 9) or **underscore**(_).
3. Identifier name cannot be **keyword**.
4. Only first **31 characters** are significant.

Valid, Invalid Identifiers

1) demo	Valid	12) xyz123	Valid
2) A		13) part#2	Invalid
3) Age	Valid	14) "char"	
4) void	Valid	15) #include	Invalid
5) MAX-ENTRIES		16) This_is_a_	
6) double	Reserved word	17) _xyz	Invalid
7) time	Invalid	18) 9xyz	Valid
8) G		19) main	Valid
9) Sue's	Reserved word	20) mutable	
10) return	Valid	21) double	Invalid
11) cout	Valid	22) max?out	Standard identifier
			Reserved word
			Reserved word
			Invalid

Constants / Literals

- Constants in C++ refer to **fixed values** that do not change during execution of program.



C++ Operators

C++ Operators

- All C language operators are valid in C++.
 1. Arithmetic operators (+, -, *, /, %)
 2. Relational operators (<, <=, >, >=, ==, !=)
 3. Logical operators (&&, ||, !)
 4. Assignment operators (+=, -=, *=, /=)
 5. Increment and decrement operators (++, --)
 6. Conditional operators (?:)
 7. Bitwise operators (&, |, ^, <<, >>)
 8. Special operators ()

Arithmetic Operators

Operator	example	Meaning
+	$a + b$	Addition
-	$a - b$	Subtraction
*	$a * b$	Multiplication
/	a / b	Division
%	$a \% b$	Modulo division- remainder

Relational Operators

Operator	Meaning
<	Is less than
<=	Is less than or equal to
>	Is greater than
>=	Is greater than or equal to
==	Equal to
!=	Not equal to

Logical Operators

Operator	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT

a	b	a && b	a b
true	true		
true	false		
false	true		
false	false		

- ☐ a && b : returns false if any of the expression is false
- ☐ a || b : returns true if any of the expression is true

Assignment operator

- We assign a value to a variable using the basic assignment operator (=).
- Assignment operator stores a value in memory.
- The syntax is

```
leftSide = rightSide ;
```



Always it is a
variable identifier.



It is either a *literal* |
a *variable identifier* |
an *expression*.

Literal: ex. `i = 1;`

Variable identifier: ex. `start = i;`

Expression: ex. `sum = first + second;`

Assignment Operators (Shorthand)

Syntax:

leftSide Op= rightSide ;



It is an *arithmetic operator.*

Ex:

x=x+3;

x+=3;

Simple assignment operator	Shorthand operator
a = a+1	a += 1
a = a-1	a -= 1
a = a * (m+n)	a *= m+n
a = a / (m+n)	a /= m+n
a = a % b	a %= b

Increment and Decrement Operators

- **Increment ++**

The ++ operator used to increase the value of the variable by **one**

- **Decrement --**

The -- operator used to decrease the value of the variable by **one**

Example:

```
x=100;  
x++;
```

After the execution the value of x will be 101.

Example:

```
x=100;  
x--;
```

After the execution the value of x will be 99.

Pre & Post Increment operator

Operator	Description
Pre increment operator (<code>++x</code>)	value of x is incremented before assigning it to the variable on the left

```
x = 10 ;  
p = ++x;
```



First increment value of
x by one

After execution
x will be **11**
p will be **11**

Operator	Description
Post increment operator (<code>x++</code>)	value of x is incremented after assigning it to the variable on the left

```
x = 10 ;  
p = x++;
```



First assign value of x

After execution
x will be **11**
p will be **10**

What is the output of this program?

```
#include <iostream>
using namespace std;
int main ()
{
    int x, y;
    x = 5;
    y = ++x * ++x;
    cout << x << y;
    x = 5;
    y = x++ * ++x;
    cout << x << y;
}
```

- (A) 749735
- (B) 736749
- (C) 367497
- (D) none of the mentioned

Conditional Operator

Syntax:

exp1 ? exp2 : exp3

Working of the ? Operator:

- **exp1** is evaluated first
 - if **exp1** is true(nonzero) then
 - **exp2** is evaluated and its value becomes the value of the expression
 - If **exp1** is false(zero) then
 - **exp3** is evaluated and its value becomes the value of the expression

Ex:

```
m=2;  
n=3;  
r=(m>n) ? m : n;
```



Value of **r** will be 3

Ex:

```
m=2;  
n=3;  
r=(m<n) ? m : n;
```



Value of **r** will be 2

Bitwise Operator

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
<<	Shift left
>>	Shift right

Bitwise Operator Examples

8 = 1000 (In Binary)
6 = 0110 (In Binary)

Bitwise & (AND)

```
int a=8,b=6,c;  
c = a & b;  
cout<<"Output ="<< c;
```

Output = 0

Bitwise | (OR)

```
int a=8,b=6,c;  
c = a | b;  
cout<<"Output ="<< c;
```

Output = 14

Bitwise << (Shift Left)

```
int a=8,b=6,c;  
c = a << 1;  
cout<<"Output ="<< c;
```

Output = 16

left shifting is the equivalent of multiplying **a** by a power of two

Bitwise >> (Shift Right)

```
int a=8,b=6,c;  
c = a >> 1;  
cout<<"Output ="<< c;
```

Output = 4

right shifting is the equivalent of dividing **a** by a power of two

New Operators in C++

::	Scope Resolution	It allows to access to the global version of variable
::*	Pointer-to-member declarator	Declares a pointer to a member of a class
->*	Pointer-to-member operator	To access pointer to class members
.*	Pointer-to-member operator	To access pointer to data members of class
new	Memory allocation operator	Allocates memory at run time
delete	Memory release operator	Deallocates memory at run time
endl	Line feed operator	
setw	Field width operator	It is a manipulator causes a linefeed to be inserted It is a manipulator specifies a field width for printing value

Scope Resolution Operator

Scope Resolution Operator(::)

```
....  
....  
{  
    int x=10;  
    ....  
    {  
        int x=1;  
        ....  
    }  
    ....  
}
```

Declaration of **x** in inner block hides declaration of same variable declared in an outer block.

Therefore, in this code both variable x refers to different data.

Block-1

- In C language, value of x declared in Block-1 is not accessible in Block-2.
- In C++, using scope resolution operator (::), value of x declared in Block-1 can be accessed in Block-2.

Scope resolution

```
#include <iostream>
using namespace std;
int m=10;
int main()
{
    int m=20;
    {
        int k=m;
        int m=3;
        cout<<"we are in inner block\n";
        cout<<"k="<

## example


```

Global declaration of variable **m**

variable **m** declared , local to main

variable **m**

declared again local to inner block

Output:

we are in inner block

k=20

m=3

::m=10

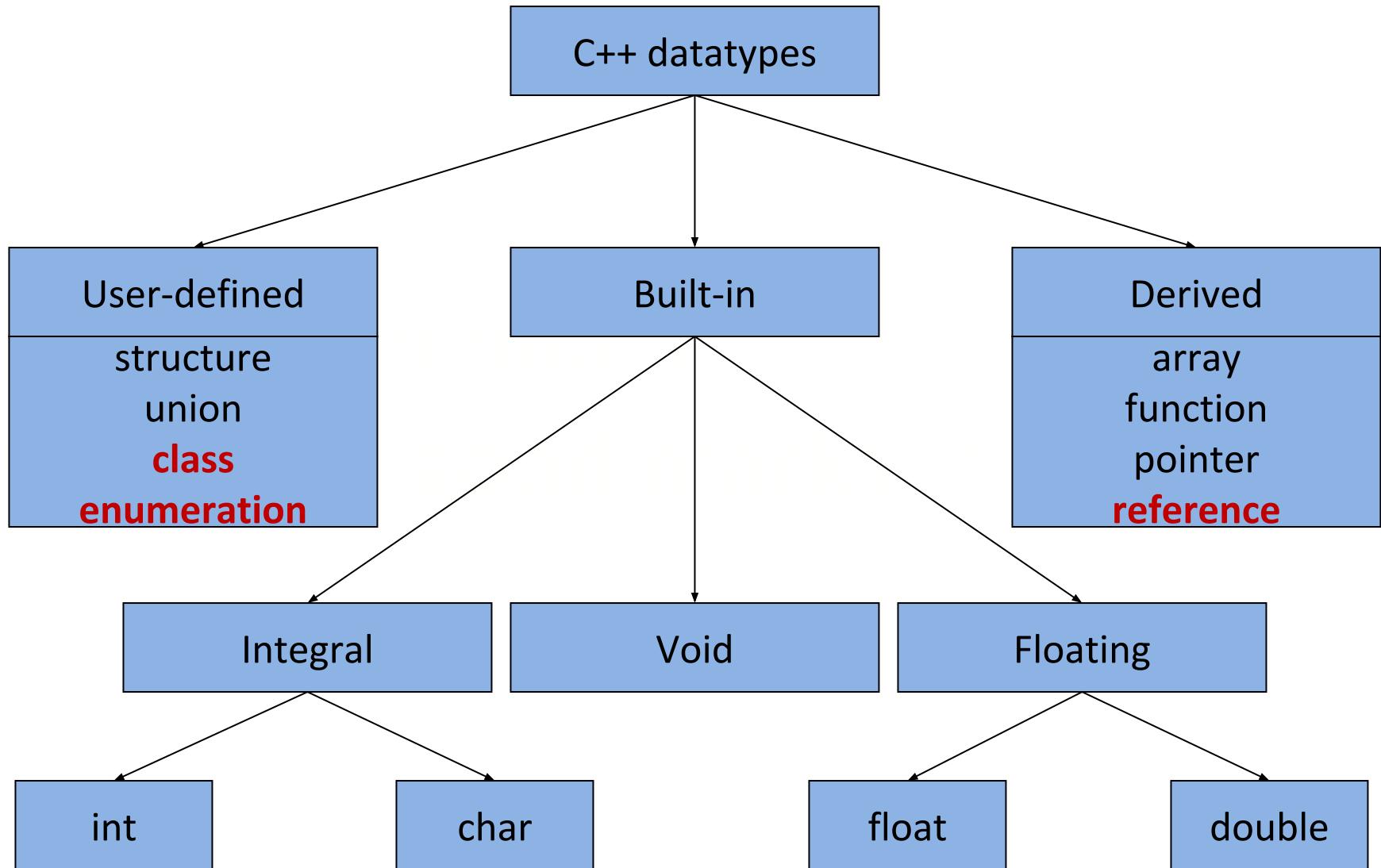
we are in outer block

m=20

::m=10

C++ Data Types

Basic Data types



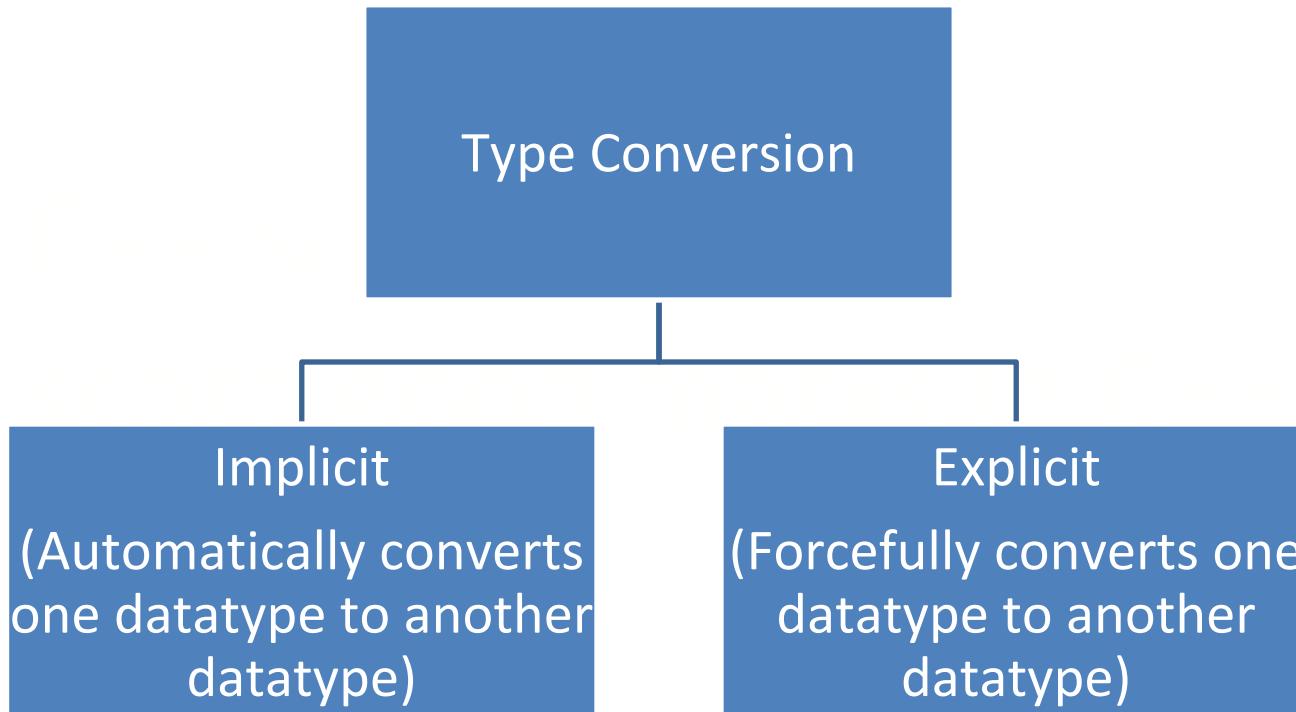
Built in Data types

Data Type	Size (bytes)	Range
char	1	-128 to 127
unsigned char	1	0 to 255
short or int	2	-32,768 to 32,767
unsigned int	2	0 to 65535
long	4	-2147483648 to 2147483647
unsigned long	4	0 to 4294967295
float	4	3.4e-38 to 3.4e+308
double	8	1.7e-308 to 1.7e+308
long double	10	3.4e-4932 to 1.1e+4932

Type Conversion

Type Conversion

- **Type Conversion** is the process of converting one predefined data type into another data type.

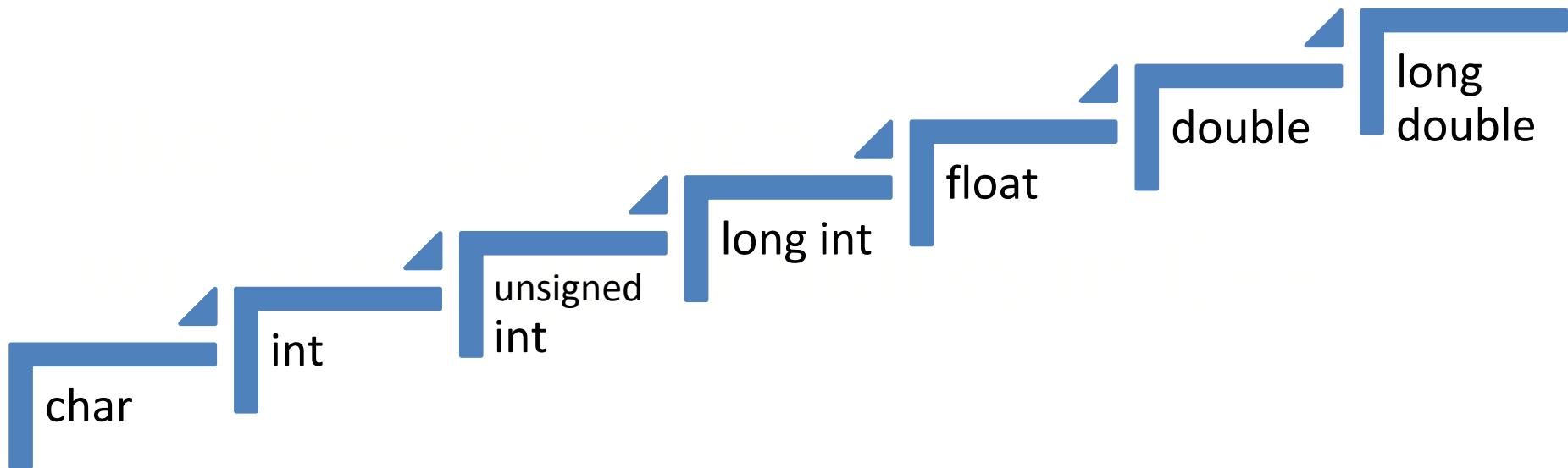


- Explicit type conversion is also known as **type casting**.

Type Conversion(Cont...)

```
int a;  
  
double b=2.55;  
  
a = b; // implicit type conversion  
  
cout << a << endl; // this will print 2  
  
a = int(b); //explicit type conversion  
  
cout << a << endl; // this will print 2
```

Implicit type conversion hierarchy



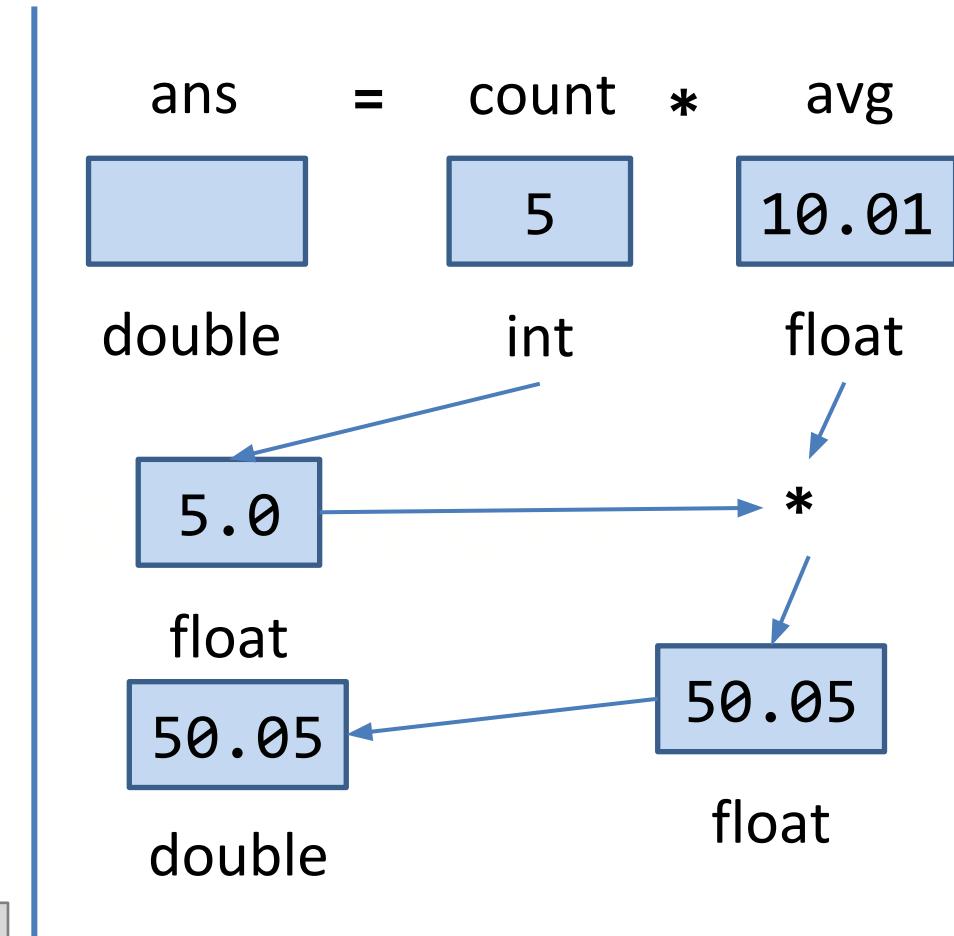
Implicit Type Conversion

```
#include <iostream>
using namespace std;
int main()
{
    int count = 5;
    float avg = 10.01;
    double ans;

    ans = count * avg;

    cout<<"Answer=";<<ans;
    return 0;
}

Output:
Answer = 50.05
```



Type Casting

- In C++ explicit type conversion is called **type casting**.
- Syntax

type-name (expression) //C++ notation

- Example

average = sum/(float) i; //C notation

average = sum/float (i); //C++ notation

Type Casting Example

```
#include <iostream>
using namespace std;
int main()
{
    int a, b, c;
    a = 19.99 + 11.99; //adds the values as float
                        // then converts the result to int
    b = (int) 19.99 + (int) 11.99; // old C syntax
    c = int (19.99) + int (11.99); // new C++ syntax

    cout << "a = " << a << ", b = " << b;
    cout << ", c = " << c << endl;

    char ch = 'Z';
    cout << "The code for " << ch << " is "; //print as char
    cout << int(ch) << endl; //print as int
    return 0;
}
```

Output:

a = 31, b = 30, c = 30
The code for Z is 90

Reference Variable

Reference Variable

- A **reference** provides an alias or a different name for a variable.
- One of the most important uses for references is in passing arguments to functions.

```
int a=5;
```

```
int &ans = a;
```

declares variable **a**

declares **ans** as reference to **a**

```
cout<< "a=" << a << endl;
cout<< "&a=" << &a << endl;
cout<< "ans=" << ans << endl;
cout<< "&ans=" << &ans << endl;
ans++;
cout<< "a=" << a << endl;
cout<< "ans=" << ans << endl;
```

OUTPUT

a=5

&a=0x6ffe34

ans=5

&ans=0x6ffe34

a=6

ans=6

Its necessary to
initialize the
Reference at the
time of declaration

Reference Variable(Cont...)

- C++ references allow you to create a second name for the a variable.
- **Reference variable** for the purpose of accessing and modifying the value of the **original variable** even if the second name (the reference) is located within a **different scope**.

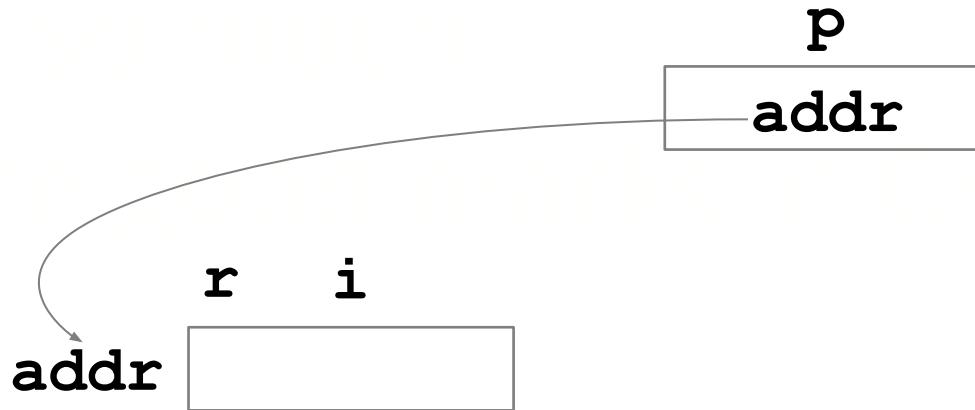
Reference Vs Pointer

References

```
int i;  
int &r = i;
```

Pointers

```
int *p = &i;
```



- A reference is a variable which **refers** to another variable.

- A pointer is a variable which **stores the address** of another variable.

Enumeration

Enumeration (A user defined Data Type)

- An **enumeration** is set of named **integer** constants.
- Enumerations are defined much like structures.

```
enum days{Sun,Mon,Tues,Wed,Thur,Fri,Sat};
```

Keyword Tag name Integer Values for symbolic constants

The diagram illustrates the structure of the enumeration code. It highlights the keyword 'enum' in red, the tag name 'days' in red, and the integer values 0 through 6. Brackets group the keyword and tag name, and another bracket groups the integer values. A curly brace at the bottom right groups all seven elements: the tag name and its corresponding integer values.

- Above statement creates **days** the name of datatype.
- By default, enumerators are assigned integer values starting with 0.
- It establishes **Sun, Mon...** and so on as symbolic constants for the integer values 0-6.

Enumeration Behaviour(Cont...)

```
enum coin { penny, nickel, dime, quarter=100,  
half_dollar, dollar};
```

The values of these symbols are

penny	0
nickel	1
dime	2
quarter	100
half_dollar	101
dollar	102

Enumeration Behaviour

```
enum days{ sun, mon, tue, wed, thu, fri, sat };  
days today; // variable today declared of type days  
  
today = tue; // Valid, because tue is an enumerator. Value 2 will  
// be assigned in today  
  
today = 6; // Invalid, because 6 is not an enumerator  
  
today++; // Invalid, today is of type days. We can not apply  
// ++ to structure variable also  
  
today = mon + fri; // Invalid  
  
int num = sat; // Valid, days data type converted to int,  
// value 6 will be assigned to num  
  
num = 5 + mon; // Valid, mon converted to int with value 1
```

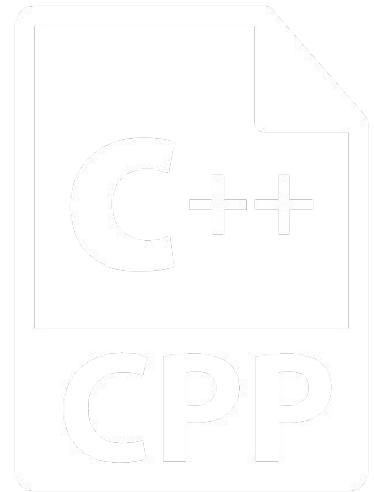
Control Structures

Control Structures

- The **if** statement:
 - Simple **if** statement
 - **if...else** statement
 - **else...if** ladder
 - **if...else** nested
- The **switch** statement :
- The **do-while** statement: An exit controlled loop
- The **while** Statement: An entry controlled loop
- The **for** statement: An entry controlled loop

Unit-2

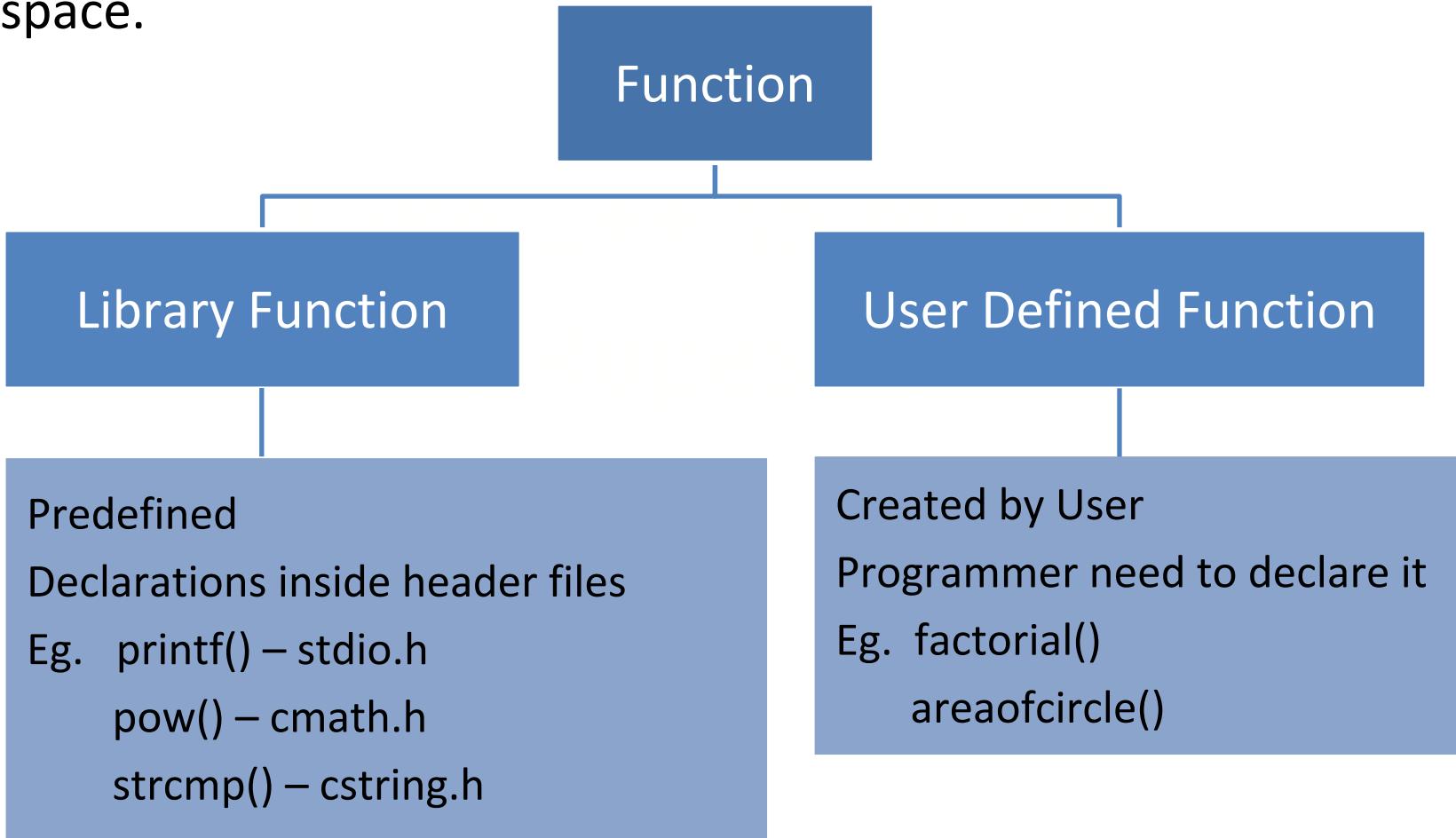
Functions



C++ Functions

C++ Function

- A **function** is a group of statements that together perform a task.
- Functions are made for code reusability and for saving time and space.



C++ Function – (Cont...)

- There are three elements of user defined function

```
void func1();  
void main()  
{  
    ....  
    func1();  
}  
void func1()  
{  
    ....  
    ....  
}
```

The diagram illustrates the three elements of a C++ function using dashed arrows and callouts:

- Function Declaration:** Points to the line `void func1();`.
- Function call:** Points to the line `func1();` within the `main()` function.
- Function definition:** Points to the block of code starting with `void func1()` and ending with a closing brace.
- Function body:** A vertical dashed bracket on the right side of the code block, spanning from the opening brace to the closing brace, indicates the scope of the function's body.

Simple Function – (Cont...)

■ Function Declaration

Syntax:

```
return-type function-name (arg-1, arg 2, ...);
```

Example: `int addition(int , int);`

■ Function Definition

Syntax:

```
return-type function-name (arg-1, arg 2, ...)
```

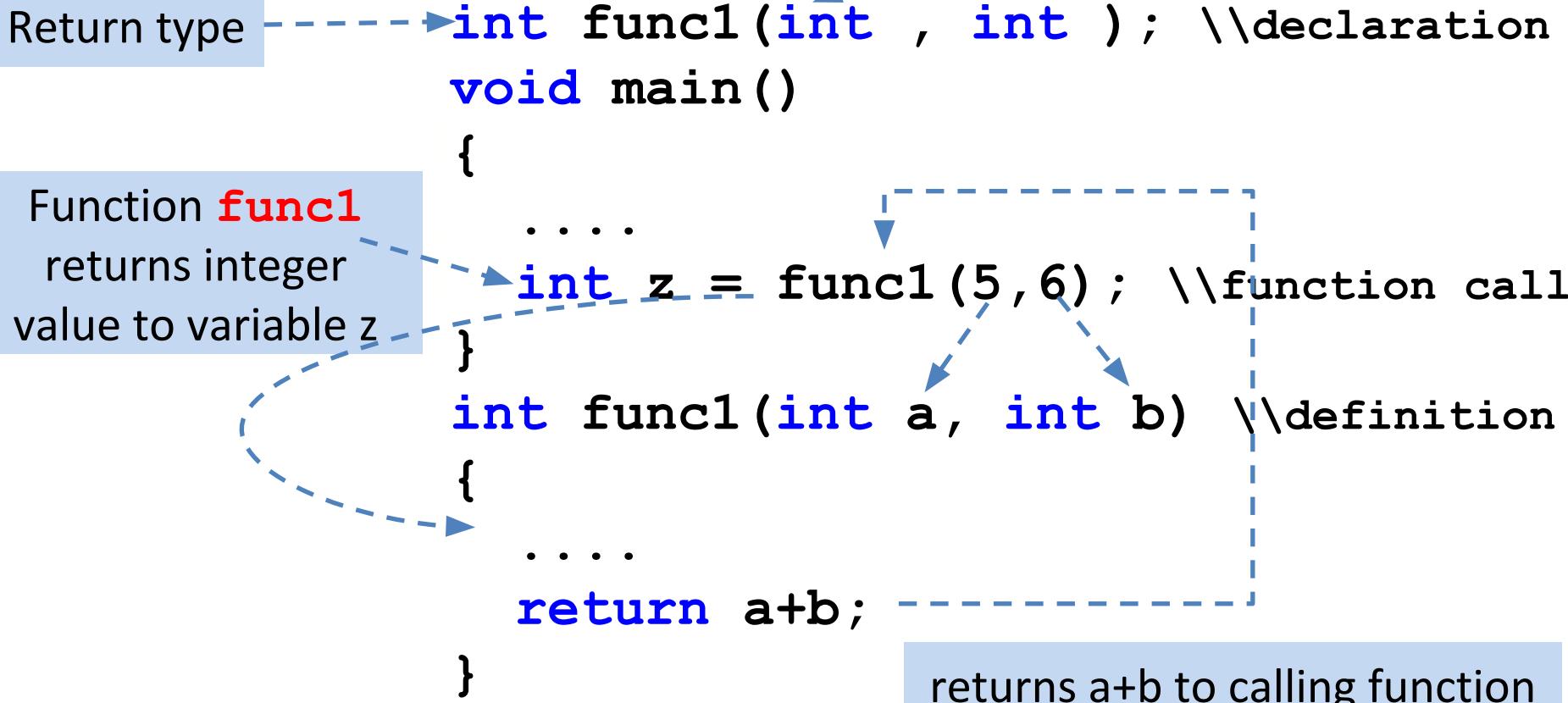
```
{  
    ... Function body  
}
```

Example: `int addition(int x, int y)`

```
{  
    return x+y;  
}
```

Categories of Function

(1) Function **with arguments** and **returns** value



Categories of Function (Cont...)

(2) Function **with arguments** but **no return** value

```
void func1(int , int ); \\function declaration
void main()
{
    ....
    func1(5,6); \\function call
}
void func1(int a, int b) \\function definition
{
    ....
    ....
}
```

Categories of Function (Cont..)

(3) Function with **no argument** but **returns** value

```
int func1();
void main()
{
    ....
    int z = func1();
}
int func1()
{
    ....
    return 99;
}
```

Categories of Function (Cont...)

(4) Function **with no argument** and **no return** value

```
void func1();  
void main()  
{  
    . . . .  
    func1();  
}  
void func1()  
{  
    . . . .  
    . . . .  
}
```

Program: Categories of function

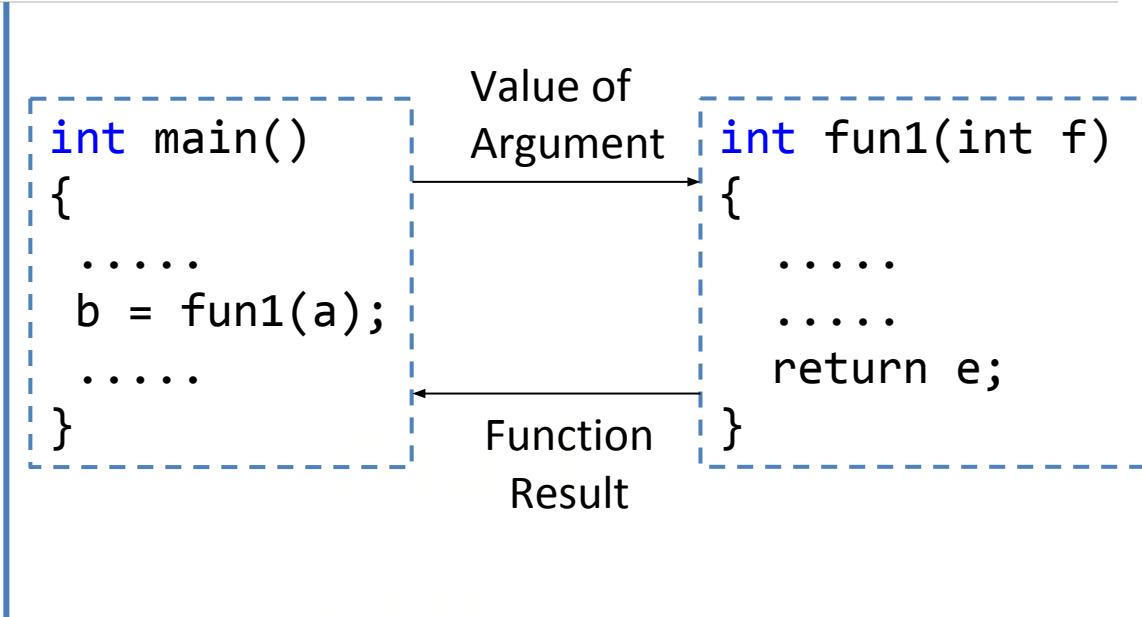
- Write C++ programs to demonstrate various categories of function, Create function **addition** for all categories.

Function with argument and returns value

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

int add(int, int);

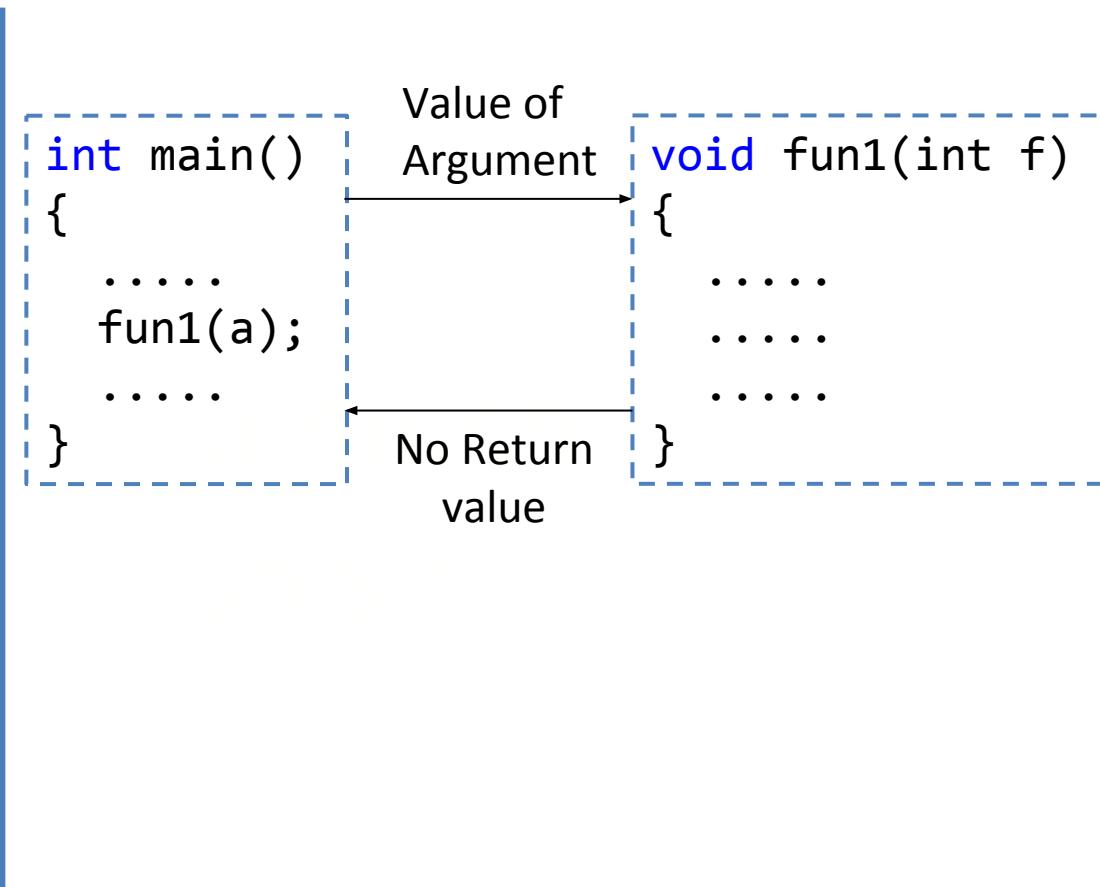
int main(){
    int a=5,b=6,ans;
    ans = add(a,b);
    cout<<"Addition is+"<<ans;
    return 0;
}
int add(int x,int y)
{
    return x+y;
}
```



Function with arguments but no return value

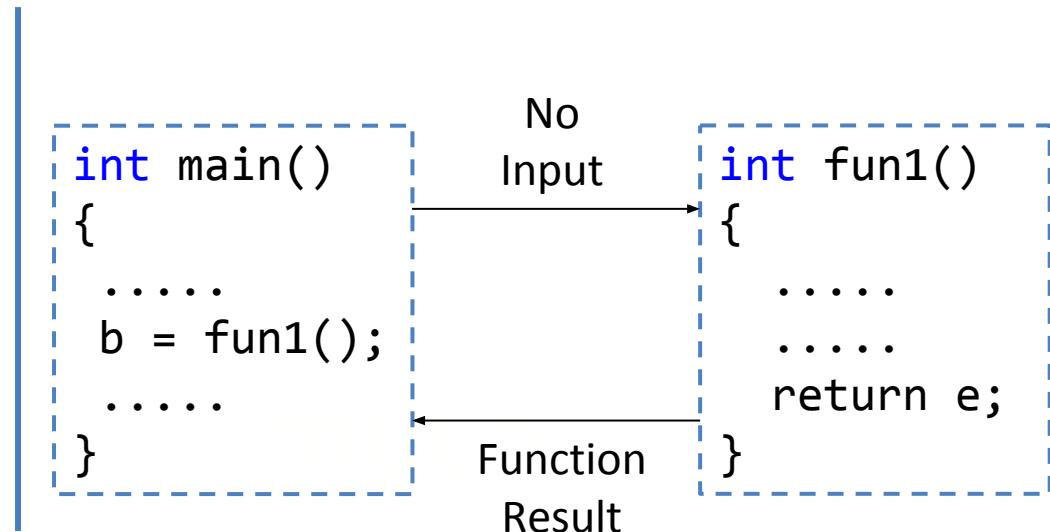
```
#include <iostream>
using namespace std;

void add(int, int);
int main()
{
    int a=5,b=6;
    add(a,b);
    return 0;
}
void add(int x,int y)
```



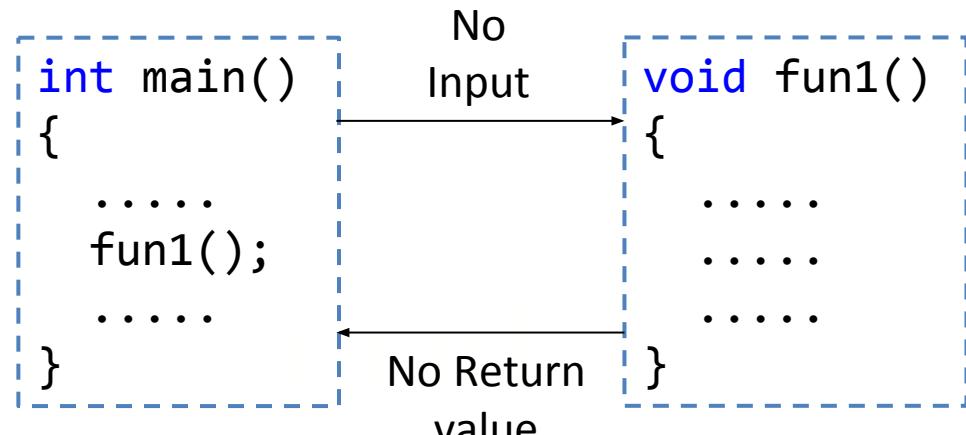
Function with no argument but returns value

```
int add();  
  
int main()  
{  
    int ans;  
    ans = add();  
    cout<<"Addition is="<<ans;  
    return 0;  
}  
void add()  
{  
    int a=5,b=6;  
    return a+b;  
}
```



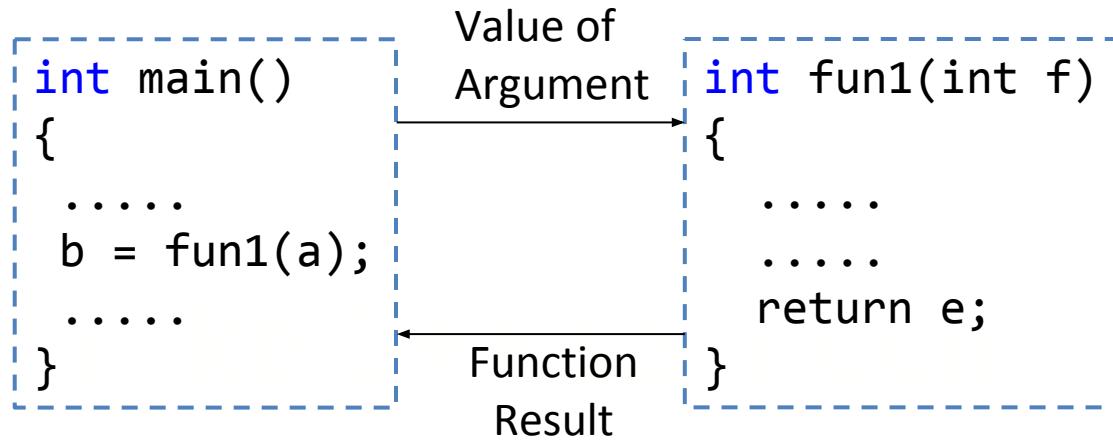
Function with no argument and no return value

```
void add();  
  
int main()  
{  
    add();  
    return 0;  
}  
  
void add()  
{  
    int a=5,b=6;  
    cout<<"Addition is+"<<a+b;  
}
```

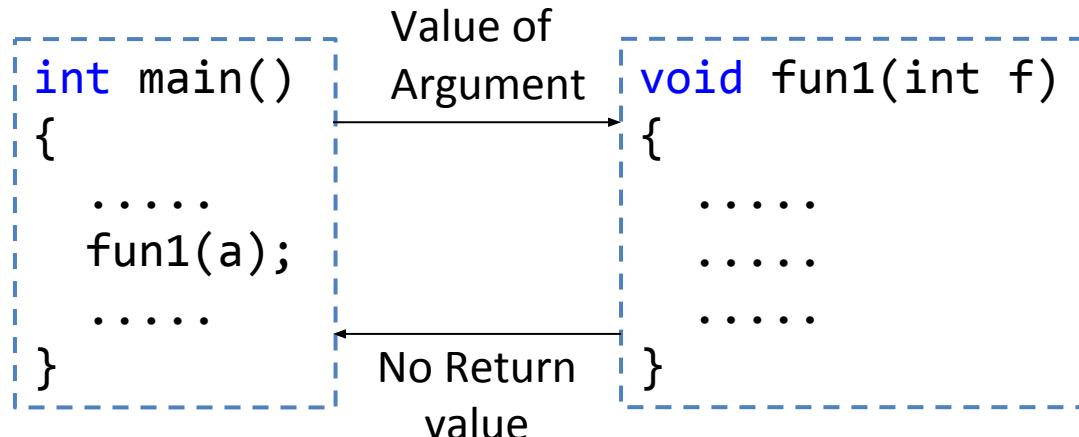


Categories of Functions Summary

(1) Function with argument and returns value

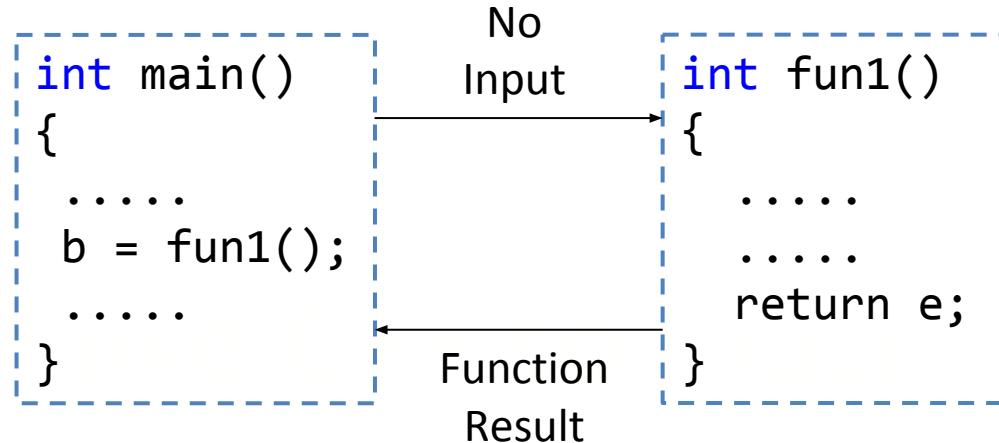


(2) Function with argument and but no return value

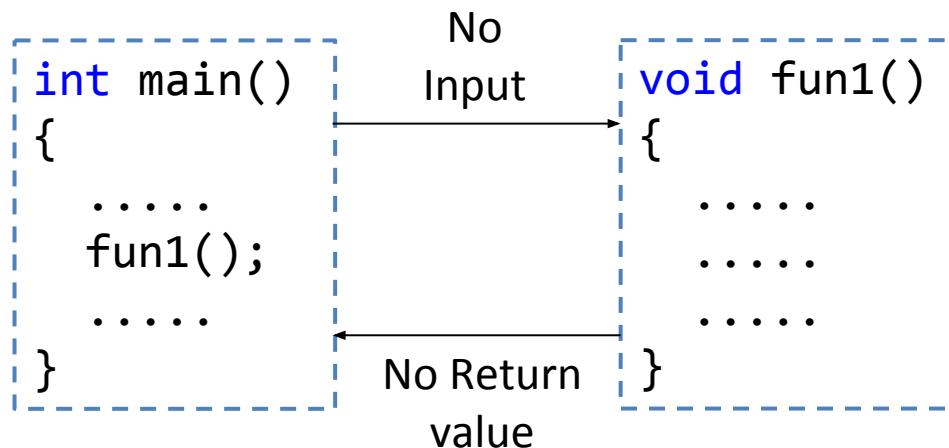


Categories of Functions Summary

(3) Function with no argument and returns value



(4) Function with no argument and but no return value



Call by Reference

pass by reference

cup = 

fillCup()

pass by value

cup = 

fillCup()

Call by reference

- The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter.
- Inside the function body, the reference is used to access the actual argument used in the call.

The diagram illustrates the mapping between actual parameters and formal parameters in a C++ function call. It consists of two columns of code. The left column shows the main function definition, and the right column shows the add function declaration. Blue arrows point from the labels to the corresponding code elements: one arrow points from 'Actual Parameters' to the arguments 'a' and 'b' in the main function, and another arrow points from 'Formal Parameters' to the parameters 'x' and 'y' in the add function.

```
int main() {  
    add(a,b);  
}  
  
void add(int x,int y) {  
    cout << x+y;  
}
```

Note:

- *Actual parameters* are parameters as they appear in function calls.
- *Formal parameters* are parameters as they appear in function declarations / definition.

Program: Swap using pointer, reference

- Write a C++ program that to swap two values using function
 1. With pass by pointer
 2. With pass by reference

Program: Solution

```
void swapptr(int *x, int *y)
{
    int z = *x;
    *x = *y;
    *y = z;
}

void swapref(int &x, int &y)
```

- Pointers as arguments
- References as arguments

```
int main()
{
    ...
    swapptr(&a,&b);
    swapref(a,b);
    ...
}
```

Program: Solution

```
void swaptr(int *, int *);  
void swapref(int &, int &);  
int main()  
{  
    int a = 45;  
    int b = 35;  
    cout<<"Before Swap\n";  
    cout<<"a=<<a<< " b=<<b<<"\n";  
  
    swaptr(&a,&b);  
    cout<<"After Swap with pass by pointer\n";  
    cout<<"a=<<a<< " b=<<b<<"\n";  
  
    swapref(a,b);  
    cout<<"After Swap with pass by reference\n";  
    cout<<"a=<<a<< " b=<<b<<"\n";  
}
```

Program: Solution (Cont...)

```
void swapptr(int *x, int *y)
{
    int z = *x;
    *x=*y;
    *y=z;
}
void swapref(int &x, int &y)
{
    int z = x;
    x = y;
    y = z;
}
```

OUTPUT
Before Swap
a=45 b=35
After Swap with pass by pointer
a=35 b=45
After Swap with pass by reference
a=45 b=35

Program: Return by Reference

- Write a C++ program to **return reference** of maximum of two numbers from function max.

Program: Solution

```
int& max(int &, int &);  
int main()  
{  
    int a=5,b=6,ans;  
    ans = max(a,b);  
    cout<<"Maximum="  
        <<ans;  
}  
int& max(int &x,int &y)  
{  
    if (x>y)  
        return x;  
    else  
        return y;  
}
```

- Function declaration returning reference

Program: Returning Reference

```
int x;  
int& setdata();  
int main()  
{  
    setdata() = 56;  
    cout<<"Value="<<x;  
    return 0;  
}  
int& setdata()  
{  
    return x;  
}
```

- `setx()` is declared with a reference type,
int&
as the return type:
- **int& setx();**
This function contains
return x;
- You can put a call to this function on the left side of the equal sign:
setx() = 92;
- The result is that the variable returned by the function is assigned the value on the right side of the equal sign.

C Preprocessors

Macros

C Preprocessors Macros

- C **Preprocessor** is a text substitution in program.
- It instructs the compiler to do pre-processing before the actual compilation.
- All **preprocessor** commands begin with a hash symbol (#).

C Preprocessor Macro Example

```
#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*r*r)
int main()
{
    int radius;
    float area;
    printf("Enter the radius: ");
    scanf("%d", &radius);
    area = circleArea(radius);
    printf("Area = %f", area);
    return 0;
}
```



Preprocessor

- Every time the program encounters circleArea(argument), it is replaced by (3.1415*(argument)*(argument)).

Inline Functions

Inline Functions

- Every time a function is called it takes a lot of extra time to execute series of instructions such as
 1. Jumping to the function
 2. Saving registers
 3. Pushing arguments into stack
 4. Returning to the calling function
- If a function body is small then overhead time is more than actual code execution time so it becomes more time consuming.
- **Preprocessor macros** is a solution to the problem of small functions in C.
- In C++, **inline function** is used to reduce the function call overhead.

Inline Functions (Cont...)

Syntax:

```
inline return-type function-name(parameters)
{
    // function code
}
```

- Add **inline** word before the function definition to convert simple function to inline function.

Example:

```
inline int Max(int x, int y)
{
    if (x>y)
        return x;
    else
        return y;
}
```

Program: Inline function

- Write a C++ program to create inline function that returns cube of given number (i.e `n=3` , `cube=(n*n*n)=27`).

Program: Solution

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

inline int cube(int s)
{
    return s*s*s;
}
int main()
{
    cout << "The cube of 3 is: " << cube(3);
    return 0;
}
```

- Calls inline function cube with argument 3

Critical situations Inline Functions

- Some of the situations inline expansion may not work
 - 1) If a **loop**, a **switch** or a **goto** exists in function body.
 - 2) If function is not returning any value.
 - 3) If function contains **static variables**.
 - 4) If function is **recursive**.

Function Overloading

Function Overloading

- Suppose we want to make functions that add 2 values, add 3 values , add 4 values

In C

```
int sum(int a, int b);
int sum(int a, int b, int c);
int sum(int a, int b, int c, int d);
```

Function with same name in a program **is not allowed** in C language

In C++

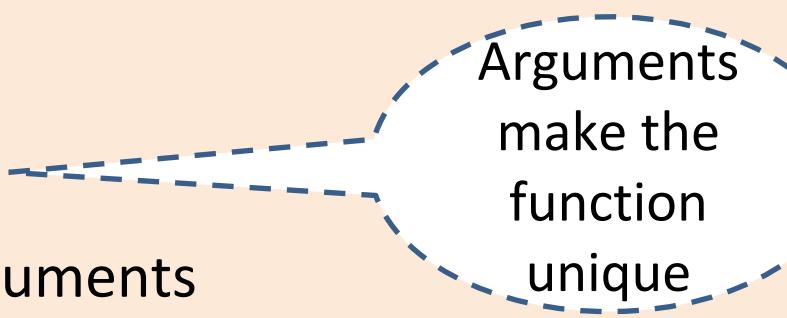
```
int sum(int a, int b);
int sum(int a, int b, int c);
int sum(int a, int b, int c, int d);
```

Function with same name in a program **is allowed** in C++ language

Function overloading – Cont...

- C++ provides **function overloading** which allows to use multiple functions **sharing the same name** .
- Function overloading is also known as **Function Polymorphism** in OOP.
- It is the practice of declaring the same function with **different signatures**.

- However, the two functions with the same name must differ in at least one of the following,
 - a) The **number** of arguments
 - b) The **data type** of arguments
 - c) The **order** of appearance of arguments
- **Function overloading** does not depends on return type.



Arguments
make the
function
unique

Function Overloading

```
int sum(int a, int b);
```

Valid

```
float sum(int a, int b);
```

Invalid

```
int sum(int a, int c);
```

Invalid

```
int sum(int a, float b);
```

Valid

```
int sum(float b, int a);
```

Valid

```
float sum(float a, float b);
```

Valid

```
int sum(int a, int b, int c);
```

Valid

```
int sum(int a, float b, int c);
```

Valid

Program: Function overloading

- Write a C++ program to demonstrate function overloading. Create function **display()** with different arguments but same name

Program: Solution (Cont...)

```
void display(int var)
{
    cout << "Integer number: " << var << endl;
}
void display(float var)
{
    cout << "Float number: " << var << endl;
}
void display(int var1, float var2) {
    cout << "Integer number: " << var1;
    cout << " and float number:" << var2;
}
```

Program: Solution

```
int main()
{
    int a = 5; float b = 5.5;
    display(a);
    display(b);
    display(a, b);
    return 0;
}
```

Program: Function overloading

- Write a C++ program to demonstrate function overloading. Create function **area()** that calculates area of circle, triangle and box.

Program #7 Solution

```
float area(int r)
{
    return 3.14*r*r;
}

float area(int h, int b)
{
    return 0.5*h*b;
}

float area(int l, int w, int h)
{
    return l*w*h;
}

int main(){
    cout<<"area of circle="<<area(5);
    cout<<"\n area of triangle="<<area(4,9);
    cout<<"\n area of box="<<area(5,8,2);
    return 0;
}
```

Default Function Arguments

Default Function Argument

Price :

5%

Discount:

SAVE

Price :

20%

Discount:

SAVE

```
int cubevolume(int l=5, int w=6, int h=7)
{
    return l*w*h;
}
```

```
int main()
{
cubevolume();
cubevolume(9);
cubevolume(15,12);
cubevolume(3,4,7);
}
```

Here, the argument is not specified for function compiler looks at its description to see how many arguments a function uses and alert program to use default values

Default Argument Example

```
int volume(int l=5,int w=6, int h=7)
{
    return l*w*h;
}
int main() {
    → cout<<"volume="<<volume()<<endl;
    → cout<<"volume="<<volume(9)<<endl;
    → cout<<"volume="<<volume(15,2)<<endl;
    → cout<<"volume="<<volume(3,4,7)<<endl;
    return 0;
}
```

- Function call passing all arguments.
- Explicitly value **3, 4, 7** passed to **l, w, h** respectively.
- Default value **7** considered for **h** respectively.

Default Arguments

- while invoking a function If the argument/s are not passed then, the default values are used.
- We must add default arguments from right to left.
- We cannot provide a default value to a particular argument in the middle of an argument list.
- Default arguments are useful in situations where some arguments always have the same value.

```
int cubevolume( int l      , int w = 2, int h      )  
{  
    return l*w*h;  
}
```



Default Arguments (Cont...)

- Legal and illegal default arguments

void f(int a, int b, **int** c=0); **Valid**

void f(int a, int b=0, **int** c=0); **Valid**

void f(int a=0, int b, **int** c=0); **Invalid**

void f(int a=0, int b, **int** c); **Invalid**

void f(int a=0, int b=0, **int** c=0); **Valid**

Common Mistakes

(1) `void add(int a, int b = 3, int c, int d = 4);`

- You cannot miss a default argument in between two arguments.
- In this case, **c** should also be assigned a default value.

(2) `void add(int a, int b = 3, int c, int d);`

- If you want a single default argument, make sure the argument is the last one.

Program: Default Arguments

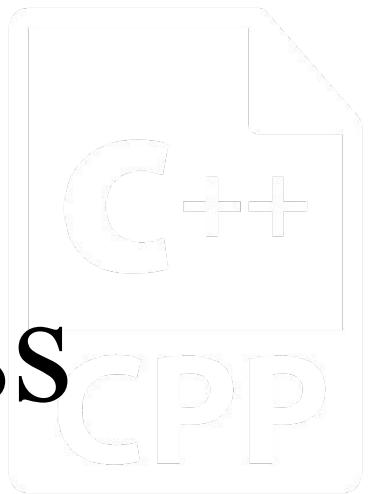
- Write a C++ program to create function **sum()**, that performs addition of 3 integers also demonstrate Default Arguments concept.

Program: Default Arguments

```
#include <iostream>
using namespace std;
int sum(int x, int y=10, int z=20)
{
    return (x+y+z);
}
int main()
{
    cout << "Sum is : " << sum(5) << endl;
    cout << "Sum is : " << sum(5,15) << endl;
    cout << "Sum is : " << sum(5,15,25) << endl;
    return 0;
}
```

Unit-2

Objects and Class



Outline

- Basics of Object and Class in C++
- Private and Public Members
- Static data and Function Members
- Constructors and their types
- Destructors
- Operator Overloading
- Type Conversion

Object and Class in C++

What is an Object?



Pen



Board



Laptop



Bench



Projector



Bike

Physical objects...

What is an Object? (Cont...)



Gujarat Technological University Ahmedabad

SEARCH RESULT:

Name	SHAH BHAUTIKVIRENBHAI				
Enrollment No.	110280106055				
Exam Seat No.	E814875				
Exam	BE SEM 8 - Regular (MAY 2015)				
Branch	CIVL ENGINEERING				

Declared On 19 Jun 2015

SUBJECT CODE	SUBJECT NAME	GRADE	INT. GRADE	ABSENT	BACKLOG
180601	Design Of Hydraulic Structures	BC	N	N	N - N - N - N
180602	Dock Harbour & Airport Engineering	BB	N	N	N - N - N - N
180603	Professional Practice & Valuation	BB	N	N	N - N - N - N
180604	Structural Design-II	BC	N	N	N - N - N - N
180605	Project -II	AA	N	N	N - N - N - N
180607	Repairs & Rehabilitation Of Structures	BB	N	N	N - N - N - N

Current Sem Backlog: 0 Total Backlog: 0 SPI: 8.20 CPI: 7.58 CGPA: 7.98

Backlog : Sem-1: 0 | Sem-2: 0 | Sem-3: 0 | Sem-4: 0 | Sem-5: 0 | Sem-6: 0 | Sem-7: 0 | Sem-8: 0 |

Online Re-Check/Re-Assessment: from 19-06-2015 to 24-06-2015 Students Guid
please send recheck query to respected department [BE,BPharm,PDOC,IMR – be@gtu.edu.in] [Diploma, DIPPharm – diploma@gtu.edu.in] [ME,MPH,MBA,MCA – pg@gtu.edu.in]. Rules of Reassessment

Note : This is a computer generated mark-sheet. Printed On : Friday, June 19, 2015 - 2:53:26 PM

Congratulation!! You have passed this exam.

Result



Bank Account

Logical objects...

Attributes and Methods of an Object



Object: Person

Attributes

Name
Age
Weight

Methods

Eat
Sleep
Walk



Object: Car

Attributes

Company
Color
Fuel type

Methods

Start
Drive
Stop



Bank Account

Object: Account

Attributes

AccountNo
HolderName
AccountType

Methods

Deposit
Withdraw
Transfer

Class

A Class is a blueprint of an object

A Class describes the object

Class car



Class: Car

Class: Car



Properties (Describe)

Company

Model

Color

Mfg. Year

Price

Fuel Type

Mileage

Gear Type

Power Steering

Anti-Lock braking system

Methods (Functions)

Start

Drive

Park

On_break

On_lock

On_turn

Objects of Class Car



Honda City



Hyundai i20



Sumo Grand



Mercedes E class



Swift Dzire

Class in C++

- A **class** is a **blueprint** or **template** that describes the object.
- A class specifies the attributes and methods of objects.

Example:

```
class car
{
    // data members and member functions
}car1;
```

- In above example class name is **car**, and **car1** is object of that class.

Specifying Class



How to declare / write class ?



How to create an object
(instance/variable of class)?



How to access class members ?

How to declare / write class ?

Class

```
class car
{
    private:
        int price;
        float mileage;
    public:
        void start();
        void drive();
};
```



Car

Attributes

Price
Mileage

Methods

Start
Drive

How to create an object ?

Syntax:

```
className objectVariableName;
```

Class

```
class car
{
    private:
        int price;
        float mileage;
    public:
        void start();
        void drive();
};
```

Object

```
int main()
{
    car c1;
    c1.start();
}
```

Object in C++

- An **object** is an instance of a class
- An **object** is a variable of type class

Class

```
class car
{
    private:
        int price;
        float mileage;
    public:
        void start();
        void drive();
};
```

Object

```
int main()
{
    car c1;
    c1.start();
    c1.drive();
}
```

Program: class, object

- Write a C++ program to create class Test having data members mark and spi.
- Create member functions **SetData ()** and **DisplayData ()** to demonstrate class and objects.

```

#include <iostream>
using namespace std;
class Test
{
private:
    int mark;
    float spi;
public:
    → void SetData()
    {
        mark = 270;
        spi = 6.5;
    }
    → void DisplayData()
    {
        cout << "Mark= "<<mark<<endl;
        cout << "spi= "<<spi;
    }
} ;

```

Program: class, object

```

→ int main()
{
    → Test o1;
    → o1.SetData();
    → o1.DisplayData();
    → return 0;
}

```

- ~~Execution of function and creation of object o1~~
- ~~Call to function definition of type Test~~
- ~~Definition of Data ()~~

```

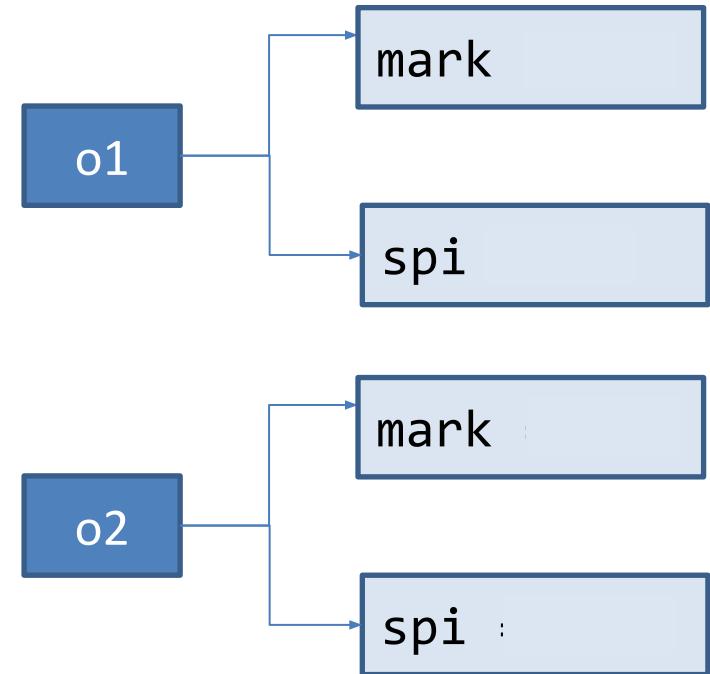
class Test
{
private:
    int mark;
    float spi;
public:
    void SetData()
    {
        cin>>mark;
        cin>>spi;
    }
    void DisplayData()
    {
        cout << "Mark= "<<mark;
        cout << "spi= "<<spi;
    }
} ;

```

```

int main()
{
    Test o1,o2;
    return 0;
}

```



Program: class, object

- Write a C++ program to create class Car having data members Company and Top_Speed.
- Create member functions **SetData()** and **DisplayData()** and create two objects of class Car.

```
class Car
{
    private:
        char company[20];
        int top_speed;
    public:
        void SetData(){
            cout<<"Enter Company:";  
            cin>>company;
            cout<<"Enter top speed:";  
            cin>>top_speed;
        }
        void DisplayData()
        {
            cout << "\nCompany:"<<company;
            cout << "\tTop Speed:"<<top_speed;
        }
};
```

Program: class, object

```
int main()
{
    Car o1;
    o1.SetData();
    o1.DisplayData();
    return 0;
```

Program: class, object

- Write a C++ program to create class Employee having data members Emp_Name, Salary, Age.
- Create member functions **SetData ()** and **DisplayData ()**.
- Create two objects of class Employee

```
class Employee
{
private:
    char name[10];
    int salary, age;

public:
    void SetData()
    {
        cin>>name>>salary>>age;
    }
    void DisplayData()
    {
        cout << "Name= "<<name<<endl;
        cout << "salary= "<<salary<<endl;
        cout << "age= "<<age;
    }
};
```

Program: class, object

```
int main()
{
    Employee o1;
    o1.SetData();
    o1.DisplayData();
    return 0;
}
```

Private and Public Members

Private



Public



Private Members

Class

```
class car
{ Private:
    long int price;
    float mileage;
    void setdata()
    {
        price = 700000;
        mileage = 18.5;
    }
};
```

- **Private** members of the class can be accessed **within the class** and from **member functions** of the class.

- A **private** member variable or function **cannot** be accessed, or even viewed from outside the class.



Private Members

- **Private** members of the class can be accessed within the class and from member functions of the class.
- They cannot be accessed outside the class or from other programs, not even from inherited class.
- If you try to access private data from outside of the class, compiler throws error.
- This feature in OOP is known as **Data hiding / Encapsulation**.
- If any other access modifier is not specified then member default acts as Private member.

Public Members

Class

```
class car
{
    private:
        long int price;
        float mileage;
    public:
        char model[10];
        void setdata()
        {
            price = 700000;
            mileage=18.53;
        }
};
```

Public members of a class can be accessed outside the class being the **object name** and dot operator '.'.

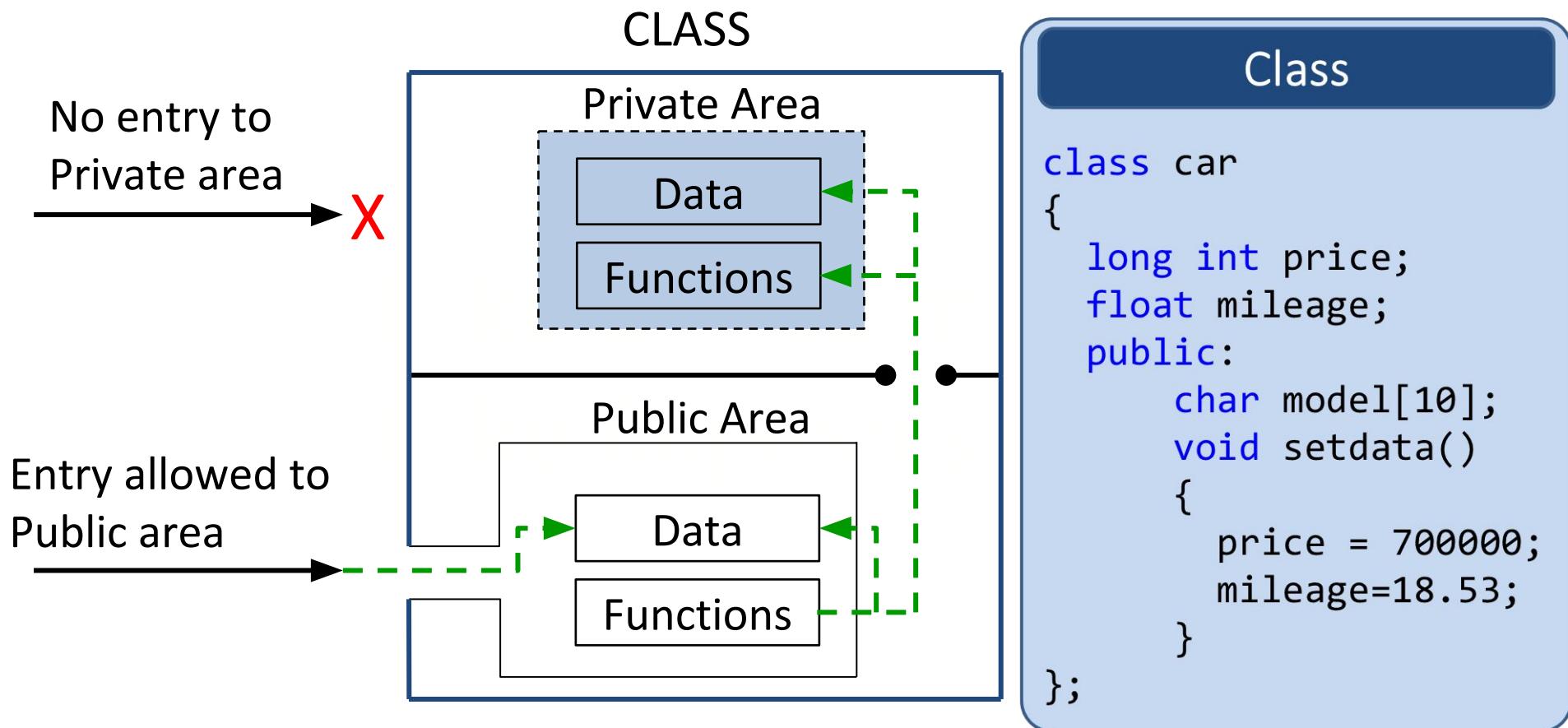
Object

```
int main()
{
    car c1;
    c1.model = "petrol";
    c1.setdata();
}
```

Public Members

- The **public** keyword makes data and functions public.
- Public members of the class are accessible by any program from anywhere.
- Class members that allow manipulating or accessing the class data are made public.

Data Hiding in Classes



Example Class in C++

```
class Test
{
    int data1;
    float data2;

public:
    void function1()
    {
        data1 = 2;
    }
    float function2()
    {
        data2 = 3.5;
        return data2;
    }
};
```

public is a
Keyword

By Default the members of a
class are **private**.

Private data and functions
can be written here

Public data and functions
can be written here

Function Definition Outside Class

Function definition outside the class

Syntax:

```
Return-type class-name :: function-name(arguments)  
{  
    Function body;  
}
```

- The membership label **class-name::** tells the compiler that the function belongs to class

Example:

```
void          SetData(int i,int j)  
{  
    mark = i;  
    spi = j;  
}
```

Function Definition outside class

```
class car
{
    private:
        float mileage;
public:
    float updatemileage();
    void setdata();

};
```

```
float car :: updatemileage()
{
    return mileage+2;
}
```

Syntax:

```
Return-type class-name :: function-name(arguments)
{
    Function body;
}
```

```
void car :: setdata()
{
    mileage = 18.5;
}
```

```
int main()
{
    car c1;
    c1.setdata();
    c1.updatemileage();
}
```

```

class Test
{
private:
    int mark;
    float spi;
public:
    void SetData(int, float);
    void DisplayData();
};

void Test :: SetData(int i, float j){
    mark = i;
    spi = j;
}
void Test :: DisplayData()
{
    cout << "Mark= " << mark;
    cout << "\nspi= " << spi;
}

```

Program: function outside class

```

int main()
{
    Test o1;
    o1.SetData(70, 6.5);
    o1.DisplayData();
    return 0;
}

```

- The membership label **Test::** tells the compiler that the **SetData()** and **DisplayData()** belongs to **Test** class

Member Functions with Arguments

Program: Function with argument

- Define class **Time** with members **hour**, **minute** and **second**.
Also define function to **setTime()** to initialize the members,
print() to display time. Demonstrate class **Time** for two
objects.

Program: Function with argument

```
#include<iostream>
using namespace std;
class Time
{
    private :
        int hour, minute, second;
    public :
        void setTime(int h, int m, int s);
        void print();
};
```

Program: Function with argument

```
void Time::setTime(int h, int m, int s)
{
    hour=h;
    minute=m;
    second=s;
}
void Time::print()
{
    cout<<"hours=\n"<<hour;
    cout<<"minutes=\n"<<minute;
    cout<<"seconds=\n"<<second;
}
```

Program: Function with argument

```
int main()
{
    int h,m,s;
    Time t1;
    cout<<"Enter hours="; cin>>h;
    cout<<"Enter minutes="; cin>>m;
    cout<<"Enter seconds="; cin>>s;

    t1.setTime(h,m,s);
    t1.print();
    return 0;
}
```

Program: Function with argument

- Define class **Rectangle** with members **width** and **height**. Also define function to **set_values()** to initialize the members, **area()** to calculate area. Demonstrate class **Rectangle** for two objects.

```
class Rectangle
{
    int width, height;
public:
    void set_values (int,int);
    int area(){
        return width*height;
    }
};

void Rectangle::set_values (int x, int y){
    width = x;    height = y;
}

int main(){
    Rectangle rect;
    rect.set_values(3,4);
    cout << "area: " << rect.area();
    return 0;
}
```

Program: Function with argument

Program: Function with argument

- Define class **Employee** with members **age** and **salary**.
 1. Also define function to **setdata()** to initialize the members.
 2. Define function **displaydata()** to display data.
- 3. Demonstrate class **Employee** for two objects.

```
int main(){  
    Employee yash,raj;  
    yash.setData(23,1500);  
    yash.displaydata();  
  
    raj.setData(27,1800);  
    raj. displaydata();  
    return 0;  
}
```

Program: Function with argument

```
class Employee{  
private :  
    int age; int salary;  
public :  
    void setData(int , int);  
    void displaydata();  
};  
void Employee::setData(int x, int y){  
    age=x;  
    salary=y;  
}  
void Employee::displaydata(){  
    cout<<"age="<<age<<endl;  
    cout<<"salary="<<salary<<endl;  
}
```

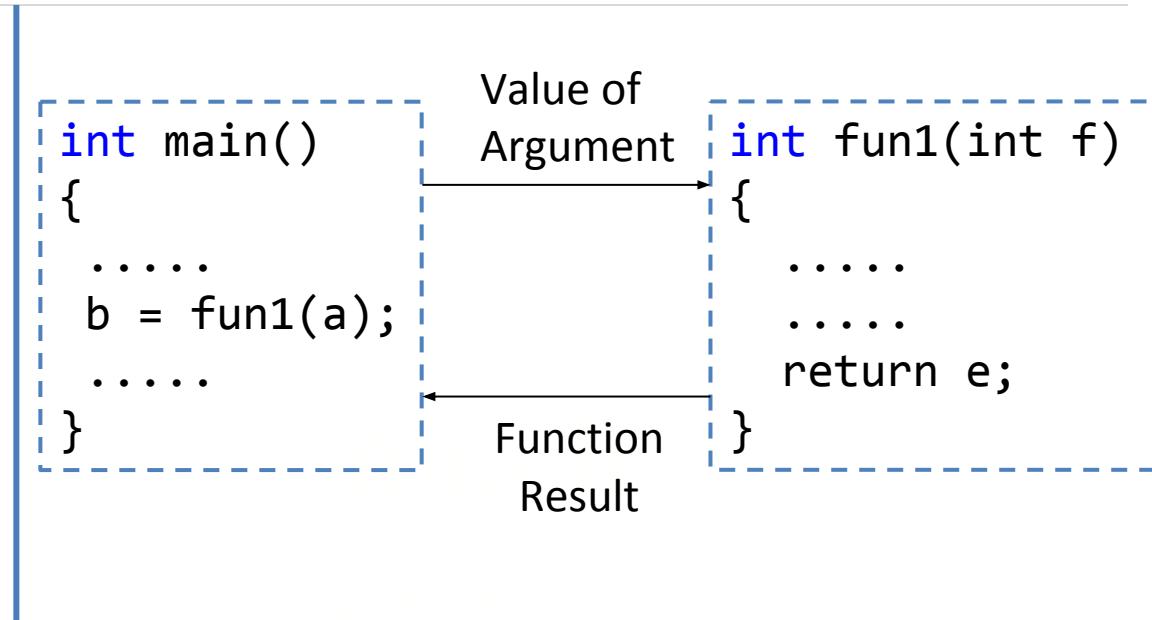
Passing Objects as Function Arguments

Function with argument and returns value

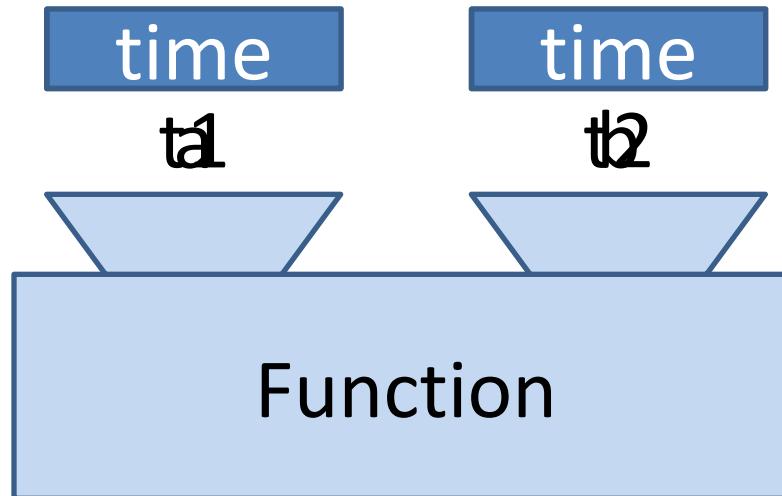
```
#include <iostream>
using namespace std;

int add(int, int);

int main(){
    int a=5,b=6,ans;
    ans = add(a,b);
    cout<<"Addition is ="<<ans;
    return 0;
}
int add(int x,int y)
{
    return x+y;
}
```



Object as Function arguments



```
void add(int x, int y)
{
    statements...
}

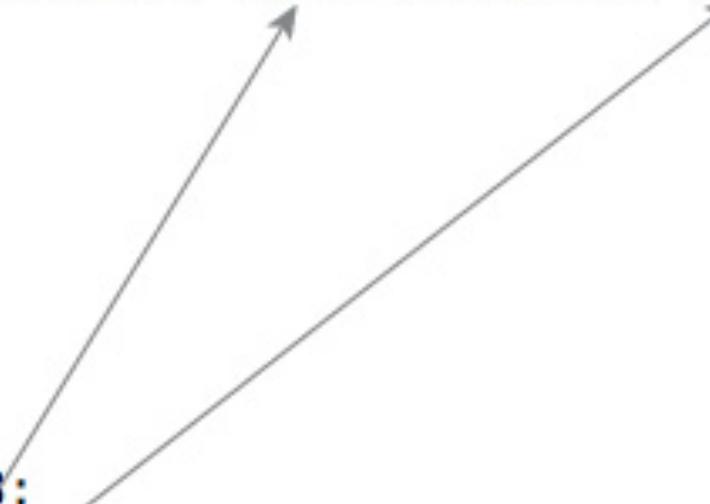
int main()
{
    int a=5,b=6;
    add(a,b);
}
```

```
void addtime(time x, time y)
{
    statements...
}

int main()
{
    time t1,t2,t3;
    t3.addtime(t1,t2);
}
```

Object as Function arguments

```
class className {  
    ...  
  
public:  
void functionName(className arg1, className arg2)  
{  
    ...  
}  
...  
};  
  
int main() {  
    className o1, o2, o3;  
    o1.functionName (o2, o3);  
}
```



```
class Time
{
    int hour, minute, second;
public :
    void getTime(){
        cout<<"\nEnter hours:";cin>>hour;
        cout<<"Enter Minutes:";cin>>minute;
        cout<<"Enter Seconds:";cin>>second;
    }
    void printTime(){
        cout<<"\nhour:"<<hour;
        cout<<"\tminute:"<<minute;
        cout<<"\tsecond:"<<second;
    }
    void addTime(Time x, Time y){
        hour = x.hour + y.hour;
        minute = x.minute + y.minute;
        second = x.second + y.second;
    }
};
```

Program: passing object as argument

Program: passing object as argument

```
int main()
{
    Time t1,t2,t3;

    t1.getTime();
    t1.printTime();

    t2.getTime();
    t2.printTime();

    t3.addTime(t1,t2);
    cout<<"\nafter adding two objects";
    t3.printTime();

    return 0;
}
```

```
t3.addTime(t1,t2);
```

Here, **hour**, **minute** and **second** represents data of object **t3** because this function is called using code **t3.addTime(t1,t2)**

Function Declaration

```
void addTime(Time x, Time y)
{
    hour = x.hour + y.hour;
    minute = x.minute + y.minute;
    second = x.second + y.second;
}
```

Program: Passing object as argument

- Define class **Complex** with members **real** and **imaginary**.
Also define function to **setdata()** to initialize the members,
print() to display values and **addnumber()** that adds two
complex objects.
- Demonstrate concept of passing object as argument.

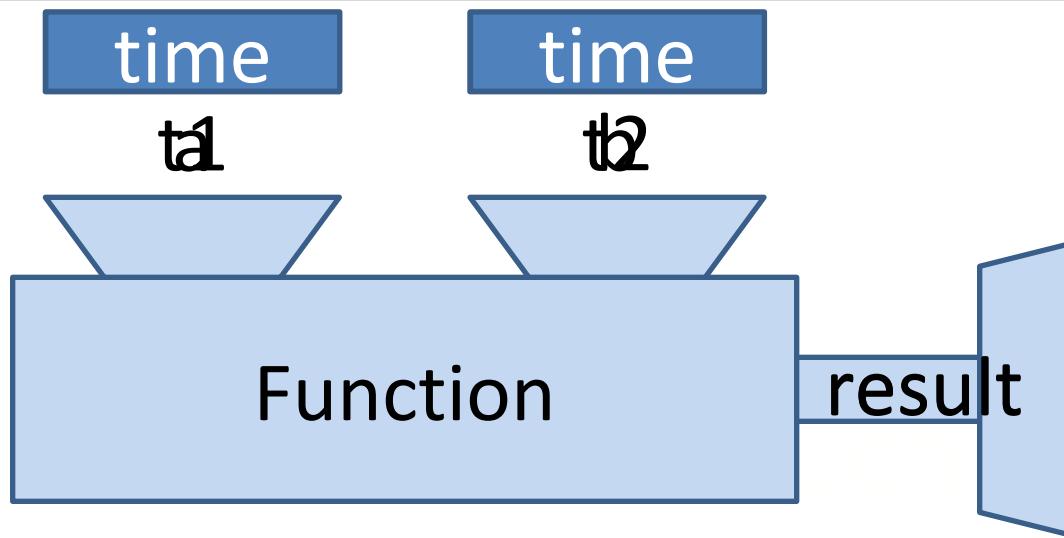
Program: Passing object as argument

```
class Complex
{
private:
    int real,imag;
public:
    void readData()
    {
        cout<<"Enter real and imaginary number:";
        cin>>real>> imag;
    }
    void addComplexNumbers(Complex comp1, Complex comp2)
    {
        real=comp1.real+comp2.real;
        imag=comp1.imag+comp2.imag;
    }
    void displaySum()
    {
        cout << "Sum = " << real<< "+" << imag << "i";
    }
};
```

```
int main()
{
    Complex c1,c2,c3;
    c1.readData();
    c2.readData();
    c3.addComplexNumbers(c1, c2);
    c3.displaySum();
}
```

Passing and Returning Objects

Passing and returning object

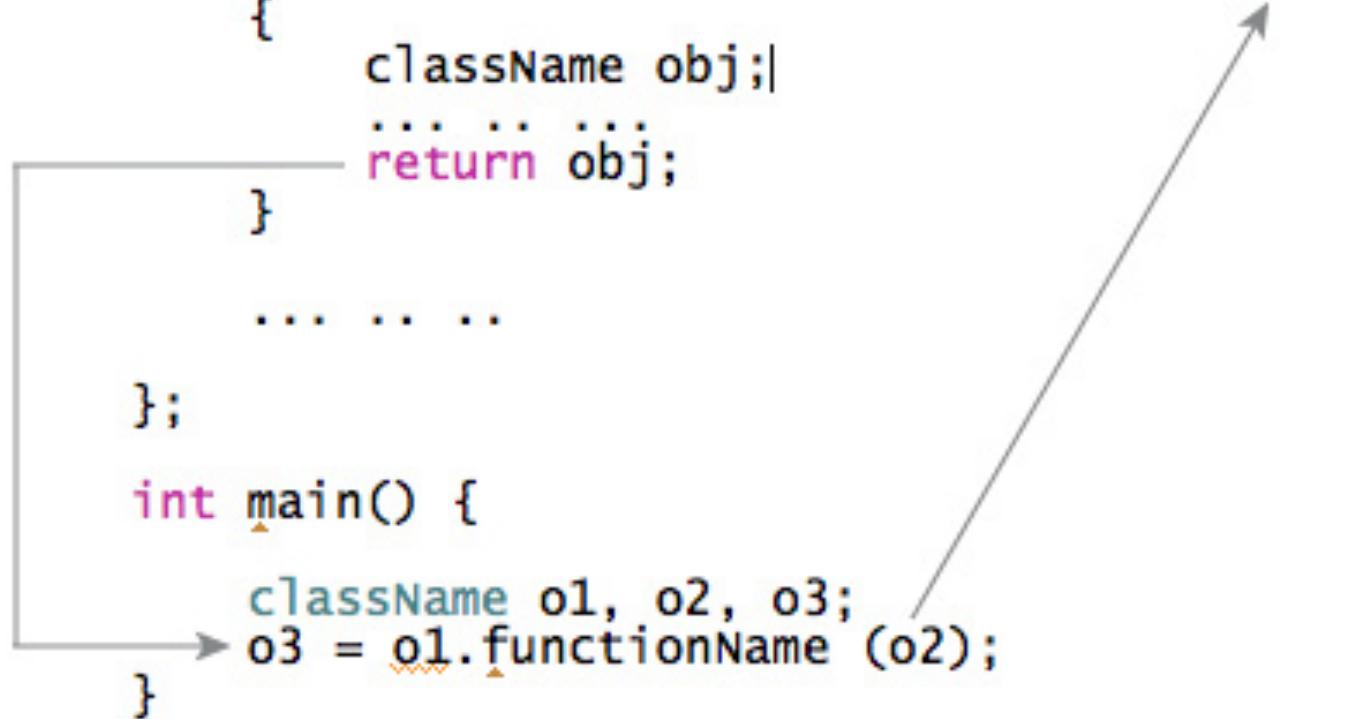


```
int add(int x, int y)
{
    return
}
int main()
{
    int a=5,b=6,result;
    result = add(a,b);
}
```

```
time addtime(time x, time y)
{
    return //object of class time
}
int main()
{
    time t1,t2,t3,result;
    result = t3.addtime(t1,t2);
}
```

Passing and returning object

```
class className {  
    ... ... ...  
public:  
    className functionName(className agr1)  
    {  
        className obj;  
        ... ... ...  
        return obj;  
    }  
    ... ... ...  
};  
  
int main() {  
    className o1, o2, o3;  
    o3 = o1.functionName (o2);  
}
```



Program: Passing and Returning an Object

- Define class **Time** with members **hour**, **minute** and **second**.
Also define function to **getTime()** to initialize the members,
printTime() to display time and **addTime()** to add two
time objects. Demonstrate class **Time**.
 1. Passing object as argument
 2. Returning object

```
class Time{  
    int hour, minute, second;  
public :  
    void getTime(){  
        cout<<"\nEnter hours:";cin>>hour;  
        cout<<"Enter Minutes:";cin>>minute;  
    }  
    void printTime(){  
        cout<<"\nhour:"<<hour;  
        cout<<"\tminute:"<<minute;  
    }  
    Time addTime(Time t1, Time t2){  
        Time t4;  
        t4.hour = t1.hour + t2.hour;  
        t4.minute = t1.minute + t2.minute;  
        return t4;  
    }  
};
```

Program: Returning
object

Program: Returning object

```
int main()
{
    Time t1,t2,t3,ans;

    t1.getTime();
    t1.printTime();

    t2.getTime();
    t2.printTime();

    ans=t3.addTime(t1,t2);
    cout<<"\nafter adding two objects";
    ans.printTime();

    return 0;
}
```

Program: Returning object

- C++ program to add two complex numbers by **Pass and Return Object** from the Function.

Program: Returning object

```
class Complex
{
private:
    int real,imag;
public:
    void readData()
    {
        cout<<"Enter real and imaginary number:";
        cin>>real>> imag;
    }
    Complex addComplexNumbers(Complex comp1, Complex comp2)
    {
        Complex temp;
        temp.real=comp1.real+comp2.real;
        temp.imag=comp1.imag+comp2.imag;
        return temp;
    }
    void displaySum()
    {
        cout << "Sum = " << real<< "+" << imag << "i";
    }
};
```

Program: Returning object

```
int main()
{
    Complex c1,c2,c3,ans;
    c1.readData();
    c2.readData();
    ans = c3.addComplexNumbers(c1, c2);
    ans.displaySum();
}
```

Nesting Member Functions

Nesting Member functions

- A member function of a class can be called by an object of that class using dot operator.
- A member function can be also called by another member function of same class.
- This is known as nesting of member functions.

```
void set_values (int x, int y)
{
    width = x;
    height = y;

    printdata();
}
```

Program: Nesting member function

- Define class **Rectangle** with member **width, height**. Also define function to **setvalue()**, **displayvalue()**. Demonstrate nested member functions.

Program: Nesting member function

```
class rectangle{
    int w,h;
public:
void setvalue(int ww,int hh)
{
    w=ww;
    h=hh;
    displayvalue();
}
void displayvalue()
{
    cout<<"width="<<w;
    cout<<"\t height="<<h;
}
};

int main(){
rectangle r1;
r1.setvalue(5,6);
r1.displayvalue();
return 0;
}
```

Memory allocation of objects

- The **member functions** are created and placed in the memory space **only once** at the time they are defined as part of a class specification.
- No separate space is allocated for member functions when the **objects** are created.
- Only space for **member variable** is allocated separately for each **object** because, the member variables will hold different data values for different objects.

Memory allocation of objects(Cont...)

Common for all objects

Member function 1

Member function 2

Memory created when, Functions defined

Object 1

Object 2

Object 3

Member variable 1

Member variable 1

Member variable 1

Member variable 2

Member variable 2

Member variable 2

Memory created when Object created

```

class Account
{
    int Account_no,Balance;
    char Account_type[10];
public:
    void setdata(int an,char at[],int bal)
    {
        Account_no = an;
        Account_type = at;
        Balance = bal;
    }
};

```

```

int main(){
    Account A1,A2,A3;
    A1.setdata(101,"Current",3400);
    A2.setdata(102,"Saving",150);
    A3.setdata(103,"Current",7900);
    return 0;
}

```

Object	A1
Account No	
Account Type	
Balance	

Object	A2
Account No	
Account Type	
Balance	

Object	A3
Account No	
Account Type	
Balance	

Static Data members / variables

Static Data members

A static data member is useful,
when all objects of the same class must **share a common information.**

Just write static keyword prefix to regular variable

It is initialized to zero when first object of class created

Only one copy is created for each object

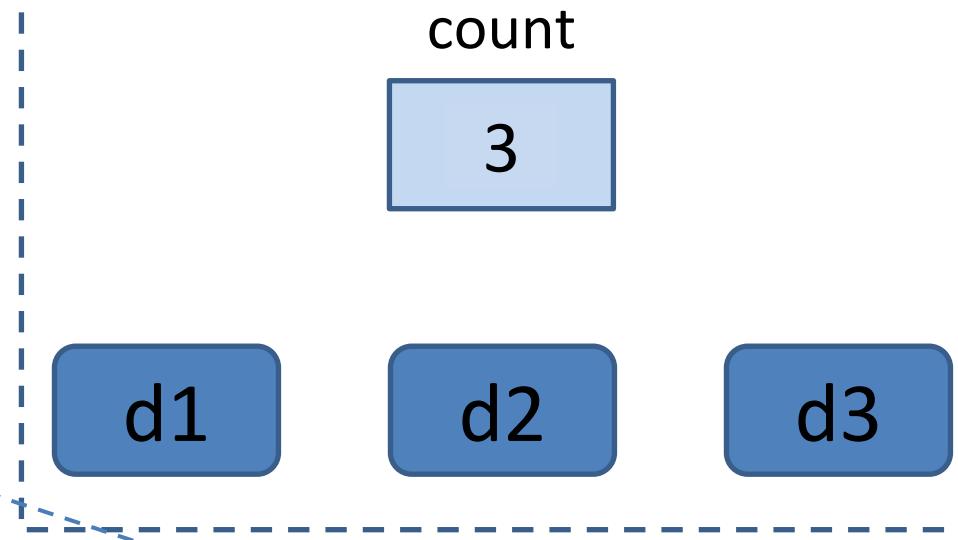
Its life time is entire program

```
class demo
{
    static int count;
public:
    void getcount()
    {
        cout<<"count="<<++count;
    }
};

int demo::count;

int main()
{
    demo d1,d2,d3;
    d1.getcount();
    d2.getcount();
    d3.getcount();
    return 0;
}
```

Static Data members



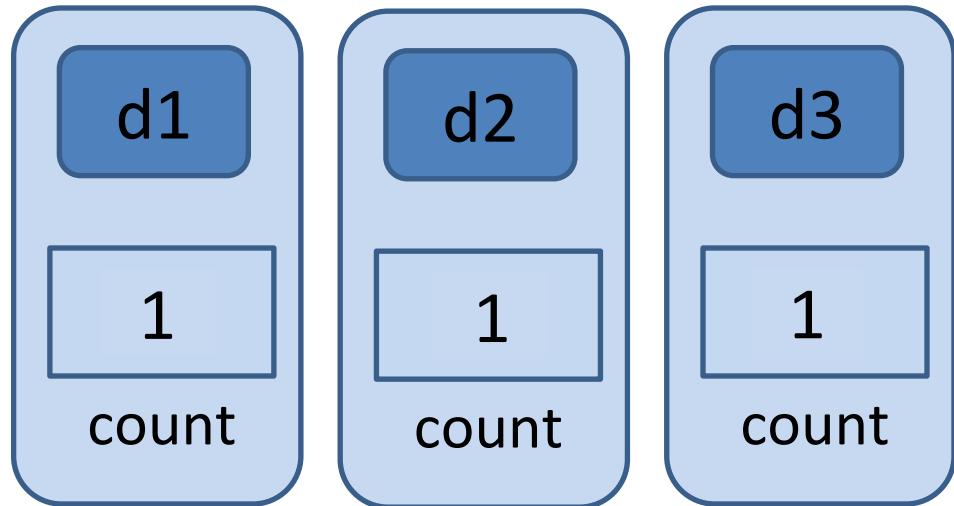
Static members are **declared** inside the class and **defined** outside the class.

Regular Data members

```
class demo
{
    int count;

public:
    void getcount()
    {
        count = 0;
        cout<<"count="<< ++count;
    }
};

int main()
{
    demo d1,d2,d3;
    d1.getcount();
    d2.getcount();
    d3.getcount();
    return 0;
}
```



Static Data Members

- Data members of the class which are shared by all objects are known as **static** data members.
- **Only one copy** of a static variable is maintained by the class and it is common for all objects.
- **Static members** are **declared** inside the class and **defined** outside the class.
- It is initialized to **zero** when the first object of its class is created.
- you cannot initialize a static member variable inside the class declaration.
- It is visible only within the class but its lifetime is the entire program.
- **Static members** are generally used to maintain values common to the entire class.

Program : Static data member

```
class item
{
    int number;
    static int count; // static variable declaration
public:
    void getdata(int a){
        number = a;
        count++; // static variable increment
    }
    void getcount(){
        cout<<"\nvalue of count: "<<count;
    }
};

int item :: count; // static variable definition
```

Program : Static data member

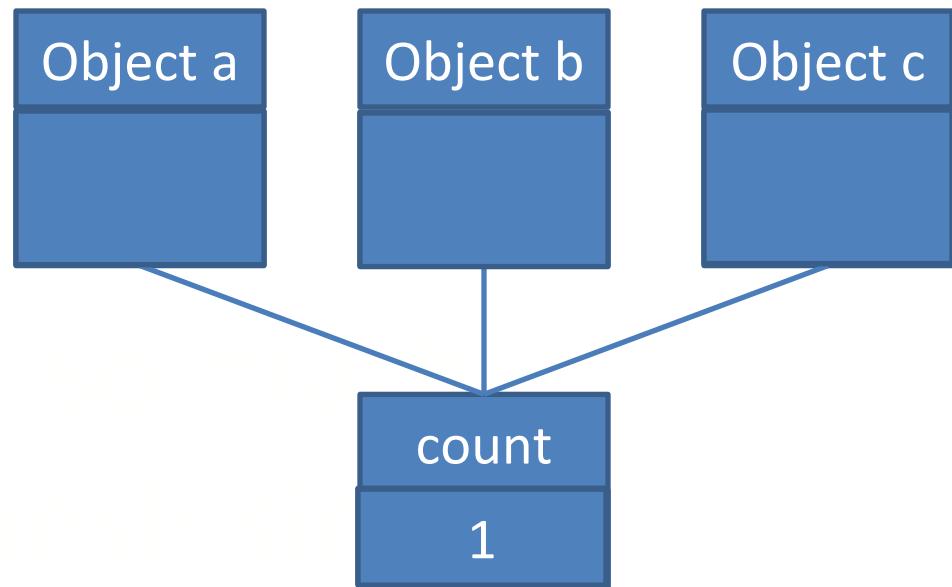
```
int main()
{
item a,b,c;

a.getdata(100);
a.getcount();

b.getdata(200);
a.getcount();

c.getdata(300);
a.getcount();

return 0;
}
```



Output:

value of count: 1
value of count: 2
value of count: 3

Program : Static data member

```
class shared {  
    static int a;  
    int b;  
public:  
    void set(int i, int j) {a=i; b=j;}  
    void show();  
};  
int shared::a;  
void shared::show()  
{  
    cout << "This is static a: " << a;  
    cout << "\nThis is non-static b: " << b; cout << "\n";  
}  
  
int main() {  
    shared x, y;  
    x.set(1, 1);  
    x.show();  
    y.set(2, 2);  
    y.show();  
    x.show();  
    return 0;  
}
```

- ~~start a variable declared outside the class~~
~~but its storage is not allocated.~~
- Storage for the variable will be allocated

Program : Static data member

```
class A
{
    int x;
public:
    A()
    {
        cout << "A's constructor called " << endl;
    }
};
```

```
class B
{
    static A a;
public:
    B()
    {
        cout << "B's constructor called " << endl;
    }
};
```

```
A B::a; // definition of a
```

```
int main()
{
    B b1, b2, b3;
    return 0;
}
```

Output:

A's constructor called
B's constructor called
B's constructor called
B's constructor called

Static Member Functions

Static Member Functions

- **Static member functions** can access only static members of the class.
- **Static member functions** can be invoked using class name, not object.
- There cannot be static and non-static version of the same function.

- They cannot be **virtual**.
- They cannot be declared as **constant** or **volatile**.
- A static member function does not have **this pointer**.

Program: Static Member function

```
class item
{
    int number;
    static int count; // static variable declaration
public:
    void getdata(int a){
        number = a;
        count++;
    }
    static void getcount(){
        cout<<"value of count: "<<count;
    }
};

int item :: count; // static variable definition
```

Program: Static Member function

```
int main()
{
    item a,b,c;

    a.getdata(100);
    item::getcount();

    b.getdata(200);
    item::getcount();

    c.getdata(300);
    item::getcount();
    return 0;
}
```

Output:

value of count: 1
value of count: 2
value of count: 3

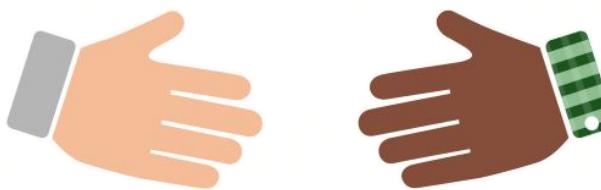
Friend Function

Friend Function

- In C++ a **Friend Function** that is a "friend" of a given class is allowed **access to private and protected data** in that class.
- A friend function is a function which is declared using **friend** keyword.

Class

```
class A
{
    private:
        int numA;
    public:
        void setA();
    friend void add();
};
```



Friend Function

```
void add()
{
    Access
    numA, numB
}
```

Class

```
class B
{
    private:
        int numB;
    public:
        void setB();
    friend void add();
};
```

Friend Function

- **Friend function** can be declared either in public or private part of the class.
- It is not a member of the class so it **cannot be called using the object**.
- Usually, it has the **objects as arguments**.

Syntax:

```
class ABC
{
    public:
        .....
        friend void xyz(argument/s); //declaration
        .....
};
```

Program: Friend Function

```
class numbers {  
    int num1, num2;  
public:  
    void setdata(int a, int b);  
    friend int add(numbers N);  
};  
void numbers :: setdata(int a, int b){  
    num1=a;  
    num2=b;  
}  
int add(numbers N){  
    return (N.num1+N.num2);  
}  
  
int main()  
{  
    numbers N1;  
    N1.setdata(10,20);  
    cout<<"Sum = "<<add(N1);  
    return 0;  
}
```

Program: Friend Function

```
class Box {  
    double width;  
public:  
    friend void printWidth( Box );  
    void setWidth( double wid );  
};  
void Box::setWidth( double wid ) {  
    width = wid;  
}  
void printWidth(Box b) {  
    cout << "Width of box : " << b.width;  
}  
int main( ) {  
Box box;  
box.setWidth(10.0);  
printWidth( box );  
return 0;  
}
```

Program: Friend Function

```
class base
{
    int val1, val2;
public:
    void get(){
        cout<<"Enter two values:";
        cin>>val1>>val2;
    }
    friend float mean(base ob);
};

float mean(base ob){
    return float(ob.val1+ob.val2)/2;
}

int main(){
    base obj;
    obj.get();
    cout<<"\n Mean value is : "<<mean(obj);
}
```

Member function, friend to another class

```
class X {  
.....  
int f();  
};  
class Y{  
.....  
friend int X :: f();  
};
```

- Member functions of one class can be made **friend function** of another class.
- The function **f** is a member of **class X** and a friend of **class Y**.

Friend function to another class

Class

```
class A
{
    private:
        int numA;
    public:
        void setA();
        friend void add();
};
```

Friend Function

```
void add()
{
    Access
    numA, numB
}
```

Class

```
class B
{
    private:
        int numB;
    public:
        void setB();
        friend void add();
};
```

Program: Friend function to another class

- Write a program to find out sum of two private data members `numA` and `numB` of two classes `ABC` and `XYZ` using a common friend function. Assume that the prototype for both the classes will be `int add(ABC, XYZ);`

Program: Friend to another class

```
class ABC {  
private:  
    int numA;  
public:  
    void setdata(){  
        numA=10;  
    }  
    friend int add(ABC, XYZ);  
};
```

```
class XYZ {  
private:  
    int numB;  
public:  
    void setdata(){  
        numB=25;  
    }  
    friend int add(ABC , XYZ);  
};
```

```
int add(ABC objA, XYZ objB){  
    return (objA.numA + objB.numB);  
}  
  
int main(){  
    ABC objA;    XYZ objB;  
    objA.setdata();  objB.setdata();  
    cout<<"Sum: "<< add(objA, objB);  
}
```

Program: Friend to another class

```
class Square; // forward declaration
class Rectangle
{
    int width=5, height=6;
public:
    friend void display(Rectangle , Square );
};

class Square
{
    int side=9;
public:
    friend void display(Rectangle , Square );
};

void display(Rectangle r, Square s)
{
    cout<<"Rectangle:"<< r.width * r.height;
    cout<<"Square:"<< s.side * s.side;
}
```

Program: Friend to another class

```
int main () {  
    Rectangle rec;  
    Square sq;  
    display(rec,sq);  
    return 0;  
}
```

Use of friend function

- It is possible to grant a nonmember function access to the private members of a class by using a **friend function**.
- It can be used to **overload binary operators**.

Constructors

What is constructor ?

A **constructor** is a block of code which is,

similar to member function

has same name as class name

called automatically when object of class created

A **constructor** is used to initialize the objects of class as soon as the object is created.

Constructor

```
class car
{
    private:
        float mileage;
    public:
        void setdata()
        {
            cin>>mileage;
        }
};
```

Same name as class name

Similar to member function

```
class car
{
    private:
        float mileage;
    public:
        car()
        {
            cin>>mileage;
        }
};
```

Called automatically on creation of object

```
int main()
{
    car c1,c2;
    c1.setdata();
    c2.setdata();
}
```

```
int main()
{
    car c1,c2;
}
```

Properties of Constructor

- **Constructor** should be declared in public section because private constructor cannot be invoked outside the class so they are useless.
- Constructors **do not have return types** and they cannot return values, not even void.

```
class car
{
    private:
        float mileage;
public:
    car()
    {
        cin>>mileage;
    }
};
```

- Constructors **cannot be inherited**, even though a derived class can call the base class constructor.
- Constructors **cannot be virtual**.
- They make implicit calls to the operators **new** and **delete** when memory allocation is required.

Constructor (Cont...)

```
class Rectangle
{
    int width,height;
public:
    Rectangle(){
        width=5;
        height=6;
        cout<<"Constructor Called";
    }
};

int main()
{
    Rectangle r1;
    return 0;
}
```

Types of Constructors

Types of Constructors

- 1) Default constructor
- 2) Parameterized constructor
- 3) Copy constructor

1) Default Constructor

- **Default constructor** is the one which invokes by default when object of the class is created.
- It is generally used to initialize the default value of the data members.
- It is also called **no argument constructor**.

```
class demo{  
    int m,n;  
public:  
    demo()  
    {  
        m=n=10;  
    }  
};
```

```
int main()  
{  
    demo d1;  
}
```

Object d1	
m	n
10	10

Program Constructor

```
class Area
{
    private:
        int length, breadth;
public:
    Area(){
        length=5;
        breadth=2;
    }
    void Calculate(){
        cout<<"\narea="<<length * breadth;
    }
};
```

```
int main(){
    Area A1;
    A1.Calculate();
    Area A2;
    A2.Calculate();
    return 0;
}
```

A1		A2	
length	breadth	length	breadth
5	2	5	2

2) Parameterized Constructor

- Constructors that can take arguments are called **parameterized constructors**.
- Sometimes it is necessary to initialize the various data elements of different objects with different values when they are created.
- We can achieve this objective by passing arguments to the constructor function when the objects are created.

Parameterized Constructor

- Constructors that can take arguments are called **parameterized constructors.**

```
class demo
{
    int m,n;
public:
    demo(int x,int y){ //Parameterized Constructor
        m=x;
        n=y;
        cout<<“Constructor Called“;
    }
};

int main()
{}
```

d1	
m	n
5	6

Program Parameterized Constructor

- Create a class **Distance** having data members **feet** and **inch**.
Create parameterized constructor to initialize members **feet** and **inch**.

3) Copy Constructor

- A **copy constructor** is used to declare and initialize an object from another object using an object as argument.
- For example:

`demo (demo &d) ; //declaration`

`demo d2 (d1) ; //copy object`

OR `demo d2=d1 ; //copy object`

- Constructor which accepts a reference to its own class as a parameter is called **copy constructor**.

3) Copy Constructor

- A **copy constructor** is used to initialize an object from another object using an object as argument.
- A Parameterized constructor which accepts a reference to its own class as a parameter is called **copy constructor**.

Copy Constructor

```
class demo
{
    int m, n;
public:
demo(int x,int y){
    m=x;
    n=y;
    cout<<"Parameterized Constructor";
}
demo(demo &x){
    m = x.m;
    n = x.n;
    cout<<"Copy Constructor";
}
};
```

```
int main()
{
    demo obj1(5,6);
    demo obj2(obj1);
    demo obj2 = obj1;
}
```

obj1 or x	
m	n
5	6

obj2	
m	n
5	6

Program: Types of Constructor

- Create a class **Rectangle** having data members **length** and **width**. Demonstrate default, parameterized and copy constructor to initialize members.

Program: Types of Constructor

```
class rectangle{  
    int length, width;  
public:  
    rectangle(){ // Default constructor  
        length=0;  
        width=0;  
    }  
    rectangle(int x, int y){ // Parameterized  
        length = x;  
        width = y;  
    }  
    rectangle(rectangle &_r){ // Copy constructor  
        length = _r.length;  
        width = _r.width;  
    }  
};
```

This is constructor overloading

Program: Types of Constructor (Cont...)

```
int main()
{
    rectangle r1; // Invokes default constructor
    rectangle r2(10,20); // Invokes parameterized
                         constructor
    rectangle r3(r2); // Invokes copy constructor
}
```

Destructor

Destructor

- **Destructor** is used to destroy the objects that have been created by a constructor.
- The syntax for **destructor** is same as that for the constructor,
 - the class name is used for the name of destructor,
 - with a **tilde (~)** sign as prefix to it.

```
class car
{
    float mileage;
public:
    car(){
        cin>>mileage;
    }
    ~car(){
        cout<<" destructor";
    }
};
```

Destructor

- never takes any argument nor it returns any value nor it has return type.
- is invoked automatically by the compiler upon exit from the program.
- should be declared in the public section.

Program: Destructor

```
class rectangle
{
    int length, width;
public:
rectangle(){ //Constructor
    length=0;
    width=0;
    cout<<"Constructor Called";
}
~rectangle() //Destructor
{
    cout<<"Destructor Called";
}
// other functions for reading, writing and
processing can be written here
};
```

```
int main()
{
    rectangle x;
    // default
    constructor is
    called
}
```

Program: Destructor

```
class Marks{  
public:  
    int maths;  
    int science;  
    //constructor  
    Marks() {  
        cout << "Inside Constructor" << endl;  
        cout << "C++ Object created" << endl;  
    }  
    //Destructor  
    ~Marks() {  
        cout << "Inside Destructor" << endl;  
        cout << "C++ Object destructed" << endl;  
    }  
};  
  
int main( )  
{  
    Marks m1;  
    Marks m2;  
    return 0;  
}
```

Operator Overloading



Operator Overloading

```
int a=5, b=10,c;  
c = a + b;
```

Operator **+** performs
addition of
integer operands a, b

```
time t1,t2,t3;  
t3 = t1 + t2;
```

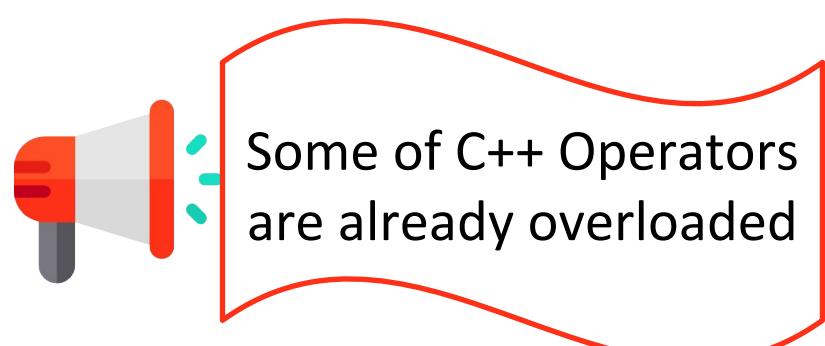
Operator **+** performs
addition of
objects of type time

```
string str1="Hello"  
string str2="Good Day";  
string str3;  
str3 = str1 + str2;
```

Operator **+** concatenates
two strings str1,str2

Operator overloading

- **Function overloading** allow you to use same function name for different definition.
- **Operator overloading** extends the overloading concept to operators, letting you assign multiple meanings to C++ operators
- **Operator overloading** giving the normal C++ operators such as +, * and == additional meanings when they are applied with **user defined data types**.



Operator	Purpose
*	As pointer, As multiplication
<<	As insertion, As bitwise shift left
&	As reference, As bitwise AND

Operator Overloading

```
int a=5, b=10,c;
```

```
c = a + b;
```

Operator + performs addition of integer
operands a, b

```
class time  
{
```

```
    int hour, minute;
```

```
};
```

```
time t1,t2,t3;
```

```
t3 = t1 + t2;
```

Operator + performs addition of objects of
type time t1,t2

```
string str1="Hello",str2="Good Day";
```

```
str1 + str2;
```

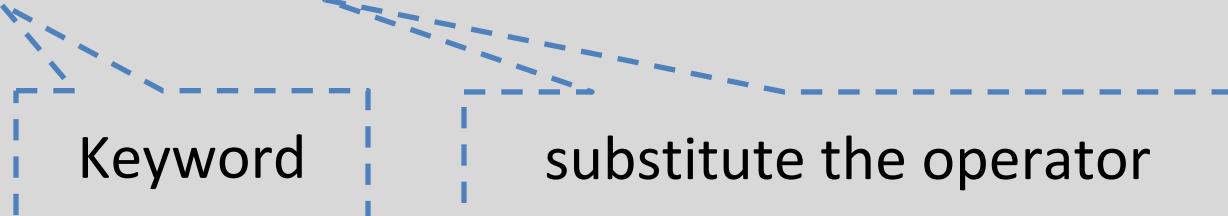
Operator + concatenates two strings
str1,str2

Operator Overloading

- Specifying more than one definition for an **operator** in the same scope, is called **operator overloading**.
- You can overload operators by creating “**operator functions**”.

Syntax:

```
return-type operator op-symbol(argument-list)
{
    // statements
}
```



Example:

```
void operator + (arguments);
int operator - (arguments);
class-name operator / (arguments);
float operator * (arguments);
```

Overloading Binary operator +

```
class complex{
    int real,imag;
public:
    complex(){}
    real=0; imag=0;
}
complex(int x,int y){
    real=x; imag=y;
}
void disp(){
    cout<<"real value="<<real<<endl;
    cout<<"imag value="<<imag<<endl;
}
complex operator + (complex);
};

complex complex::operator + (complex c){
    complex tmp;
    tmp.real = real + c.real;
    tmp.imag = imag + c.imag;
    return tmp;
}
```

```
int main()
{
    complex c1(4,6),c2(7,9);
    complex c3;
    c3 = c1 + c2;
    c1.disp();
    c2.disp();
    c3.disp();
    return 0;
}
```

Similar to function call

c3=c1.operator +(c2);

Binary Operator Arguments

```
result = obj1.operator symbol (obj2); //function notation
```

```
result = obj1 symbol obj2; //operator notation
```

```
complex operator + (complex x)
```

```
{
```

```
    complex tmp;
```

```
    tmp.real = real + x.real;
```

```
    tmp.imag = imag + x.imag;
```

```
    return tmp;
```

```
}
```

```
result = obj1.display();
```

```
void display()
```

```
{
```

```
    cout<<"Real="<<real;
```

```
    cout<<"Imaginary="<<imag;
```

```
}
```

Operator Overloading

- **Operator overloading** is compile time polymorphism.
- You can overload most of the built-in operators available in C++.

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

Operator Overloading using Friend Function

Invoke Friend Function in operator overloading

```
result = operator symbol (obj1,obj2); //function notation
```

```
result = obj1 symbol obj2; //operator notation
```

```
friend complex operator +(complex c1,complex c2)
{
    complex tmp;
    tmp.r=c1.r+c2.r;
    tmp.i=c1.i+c2.i;
    return tmp;
}
```

```
int main()
{
    complex c1(4,7),c2(5,8);
    complex c3;
    c3 = c1 + c2;
    c3 = operator +(c1,c2);
}
```

Overloading Binary operator ==

```
class complex{                                int main()
    int r,i;
public:
complex(){                                {
    r=i=0;}                                complex c1(5,3),c2(5,3);
complex(int x,int y){                      if(c1==c2)
    r=x;                                cout<<"objects are equal";
    i=y;}                                else
void display(){                                cout<<"objects are not equal";
    cout<<"\nreal=" <<r<<endl;
    cout<<"imag=" <<i<<endl;}
int operator==(complex);                    return 0;
};                                            }
int complex::operator ==(complex c){
if(r==c.r && i==c.i)
    return 1;
else
    return 0;}
```

Overloading Unary Operator

Overloading Unary operator –

```
class space {  
    int x,y,z;  
public:  
    space(){  
        x=y=z=0; }  
    space(int a, int b,int c){  
        x=a; y=b; z=c; }  
    void display(){  
        cout<<"\nx="<<x<<,y="<<y<<,z="<<z;  
    }  
    void operator-();  
};  
void space::operator-() {  
    x=-x;  
    y=-y;  
    z=-z;  
}
```

```
int main()  
{  
    space s1(5,4,3);  
    s1.display();  
    -s1;  
    s1.display();  
    return 0;  
}
```

Overloading Unary operator --

```
class space {  
    int x,y,z;  
public:  
    space(){  
        x=y=z=0; }  
    space(int a, int b,int c){  
        x=a; y=b; z=c; }  
    void display(){  
        cout<<"\nx="<<x<<,y="<<y<<,z="<<z;  
    }  
    void operator--();  
};  
void space::operator--() {  
    x--;  
    y--;  
    z--;  
}
```

```
int main()  
{  
    space s1(5,4,3);  
    s1.display();  
    --s1;  
    s1.display();  
    return 0;  
}
```

Overloading Prefix and Postfix operator

```
class demo
{
    int m;
public:
    demo(){ m = 0;}
    demo(int x)
    {
        m = x;
    }
    void operator ++()
    {
        ++m;
        cout<<"Pre Increment="<<m;
    }
    void operator ++(int)
    {
        m++;
        cout<<"Post Increment="<<m;
    }
};
```

```
int main()
{
    demo d1(5);
    ++d1;
    d1++;
}
```

Invoking Operator Function

- Binary operator

```
operand1 symbol operand2
```

- Unary operator

```
operand symbol  
symbol operand
```

- Binary operator using friend function

```
operator symbol (operand1, operand2)
```

- Unary operator using friend function

```
operator symbol (operand)
```

Rules for operator overloading

- Only existing operator can be overloaded.
- The overloaded operator must have at least one operand that is user defined type.
- We cannot change the basic meaning and syntax of an operator.

Rules for operator overloading (Cont...)

- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
- We cannot overload following operators.

Operator	Name
. and .*	Class member access operator
::	Scope Resolution Operator
sizeof()	Size Operator
?:	Conditional Operator

Type Conversion

Type Conversion

```
F = C * 9/5 + 32
```

float int

If different data types are **mixed in expression**, C++ applies automatic type conversion as per certain rules.

```
int a;  
float b = 10.54;  
a = b;
```

integer
(Basic)

float
(Basic)

a = 10;

- float is converted to integer automatically by compiler.
- basic to basic type conversion.

- An assignment operator causes automatic type conversion.
- The data type to the right side of **assignment operator** is automatically converted data type of the variable on the left.

Type Conversion

```
Time t1;  
int m;  
m = t1;
```

integer
(Basic)

Time
(Class)

Time
(Class)

integer
(Basic)

- **class type** will not be converted to **basic type** OR **basic type** will not be converted **class type** automatically.

Type Conversion

- C++ provides mechanism to perform automatic type conversion if all variable are of **basic type**.
- For user defined data type programmers have to convert it by using **constructor** or by using **casting operator**.
- Three type of situation arise in user defined data type conversion.
 1. Basic type to Class type (Using Constructors)
 2. Class type to Basic type (Using Casting Operator Function)
 3. Class type to Class type (Using Constructors & Casting Operator Functions)

(1) Basic to class type conversion

- Basic to class type can be achieved **using constructor.**

```
class sample
{
    int a;
public:
sample(){}
sample(int x){
    a=x;
}
void disp(){
    cout<<"The value of a="<<a;
}
};
```

```
int main()
{
    int m=10;
    sample s;
    s = m;
    s.disp();
    return 0;
}
```

(2) Class to basic type conversion

- The Class type to Basic type conversion is done **using casting operator function.**
- The casting operator function should satisfy the following conditions.
 1. It must be a class member.
 2. It must not mention a return type.
 3. It must not have any arguments.

Syntax:

```
operator destinationtype()
{
    ...
    return
}
```

Program: Class to basic type conversion

```
class sample
{
    float a;
public:
sample()
{
    a=10.23;
}
operator int() //Casting operator
function
{
    int x;
    x=a;
    return x;
}
};
```

```
int main()
{
    sample S;
    int y= S; //Class to Basic
               conversion
    cout<<"The value of y="<<y;
    return 0;
}
```

Explicit type conversion

$y = \text{int}(S);$

Automatic type
conversion

$y = S;$

Program: Class to basic type conversion

```
class vector{  
    int a[5];  
public:  
vector(){  
for(int i=0;i<5;i++)  
    a[i] = i*2;  
}  
operator int();  
};
```

```
vector:: operator int() {  
int sum=0;  
for(int i=0;i<5;i++)  
    sum = sum + a[i];  
return sum; }
```

```
int main()  
{  
vector v;  
int len;  
len = v;  
cout<<"Length of V="<<len;  
return 0;  
}
```

(3) Class type to Class type

- It can be achieved by two ways
 1. Using constructor
 2. Using casting operator function

```
class alpha
{
    int commona;
public:
    alpha(){}
    alpha(int x)
    {
        commona = x;
    }
    int getvalue()
    {
        return commona;
    }
};

int main()
{
    alpha obja(10);
    beta objb(obja);
    beta objb(20);
    obja = objb;
}
```

Program: Class type to Class type

```
class beta
{
    int commonb;
public:
    beta(){}
    beta(int x)
    {
        commonb = x;
    }
    beta(alpha temp) //Constructor
    {
        commonb = temp.getvalue();
    }
    operator alpha() //operator function
    {
        return alpha(commonb);
    }
};
```

```
class stock2 ;
class stock1{
    int code , item ;
    float price ;
public :
stock1 ( int a , int b , int c ) {
    code = a ; item = b ; price = c ;
}
void disp ()  {
    cout << " code " << code << "\n " ;
    cout << " items " << item << "\n " ;
    cout << " price per item Rs. " << price << "\n " ;
}
int getcode (){ return code; }
int getitem (){ return item ; }
int getprice (){ return price ; }
operator float ()  {
    return ( item*price ) ;
}
};
```

Program: Type Conversion

```
class stock2{
    int code ;
    float val ;
public :
stock2 ()  {
    code = 0; val = 0 ;
}
stock2( int x , float y ){
    code = x ; val = y ;
}
void disp ()  {
    cout << " code " << code << " \n " ;
    cout << " total value Rs. " << val << " \n " ;
}
stock2( stock1 p )  {
    code = p.getcode() ;
    val = p.getitem() * p.getprice() ;
}
};
```

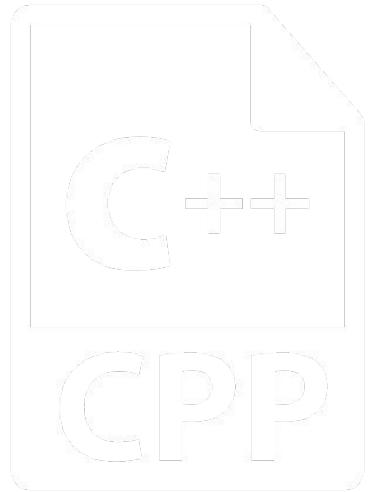
Program: Type Conversion

```
int main()
{
    stock1 i1 ( 101 , 10 ,125.0 ) ;
    stock2 i2 ;
    float tot_val = i1;
    i2 = i1 ;
    cout << " Stock Details : Stock 1 type " << " \n " ;
    i1.disp ();
    cout << " Stock Value " << " - " ;
    cout << tot_val << " \n " ;
    cout << " Stock Details : Stock 2 type " << " \n " ;
    i2.disp () ;
    return 0 ;
}
```

Program: Type Conversion

Unit-5

Inheritance



Outline

- Concept of inheritance
- Types of inheritance
 - 1. Single
 - 2. Multiple
 - 3. Multilevel
 - 4. Hierarchical
 - 5. Hybrid
- Protected members
- Overriding
- Virtual base class

Concept of Inheritance

class Doctor

Attributes:

Age, Height, Weight

Methods:

Talk()

Walk()

Eat()

Diagnose()

class Footballer

Attributes:

Age, Height, Weight

Methods:

Talk()

Walk()

Eat()

Playfootball()

class Businessman

Attributes:

Age, Height, Weight

Methods:

Talk()

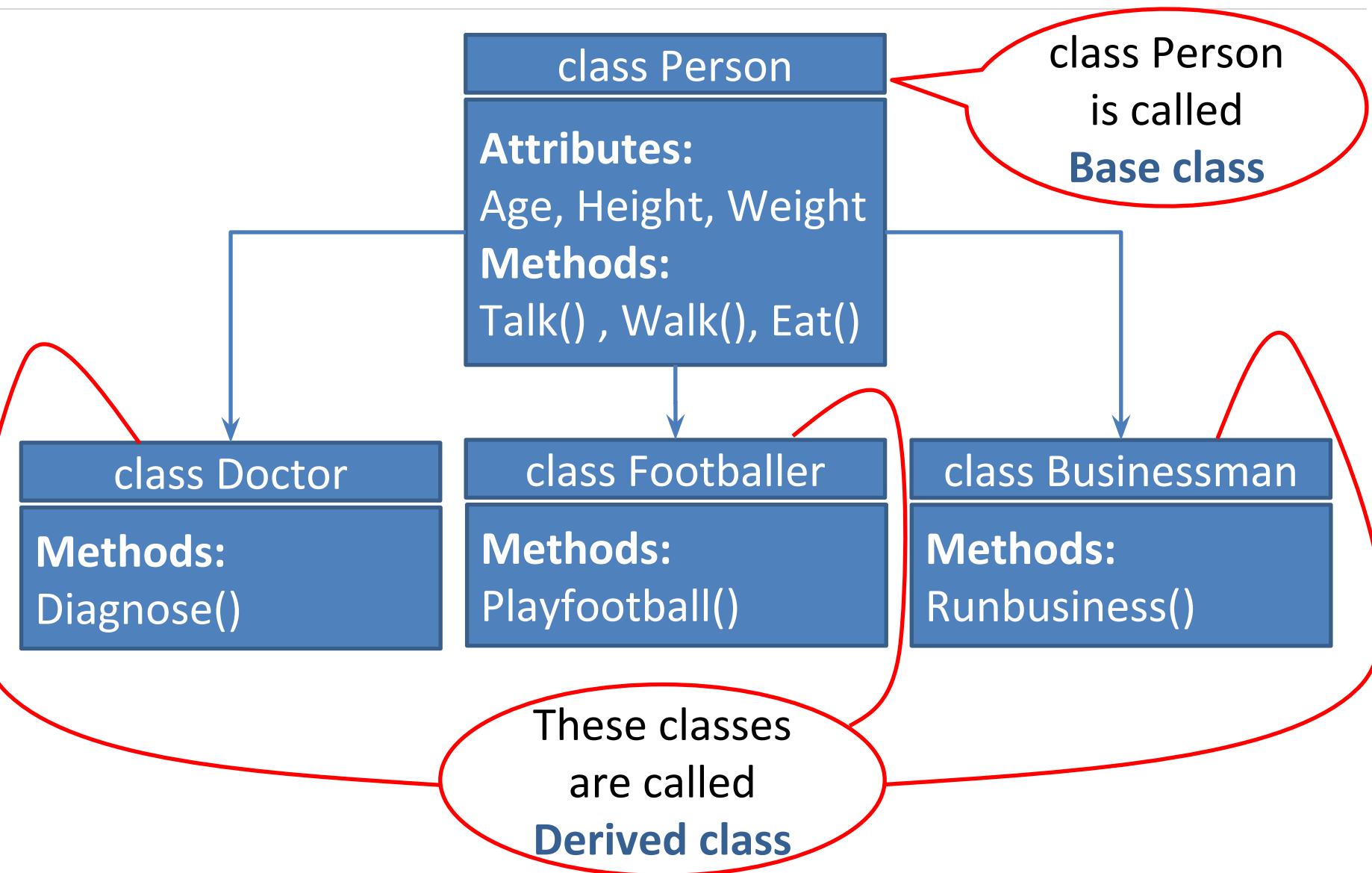
Walk()

Eat()

Runbusiness()

- All of the classes have common attributes (Age, Height, Weight) and methods (Walk, Talk, Eat).
- However, they have some special skills like Diagnose, Playfootball and Runbusiness.
- In each of the classes, you would be copying the same code for Walk, Talk and Eat for each character.

Concept of Inheritance(Cont...)



Inheritance

- **Inheritance** is the process, by which class can acquire(reuse) the properties and methods of another class.
- The mechanism of deriving a new class from an old class is called **inheritance**.
- The new class is called **derived class** and old class is called **base class**.
- The derived class may have all the features of the base class and the programmer can add new features to the derived class.

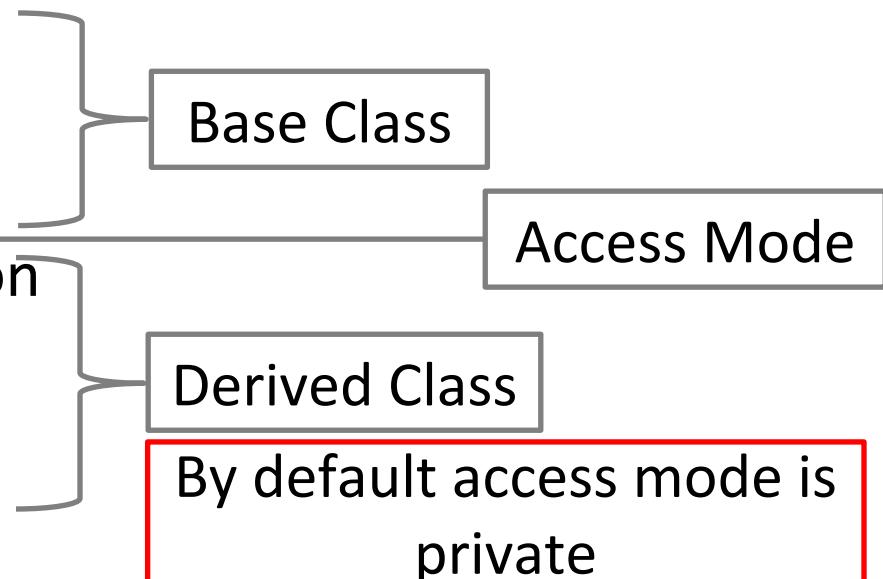
Syntax to Inherit class

Syntax:

```
class derived-class-name : access-mode  
base-class-name  
{  
    // body of class  
};
```

Example:

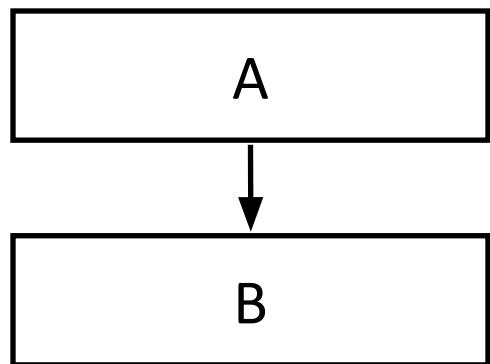
```
class person{  
    //body of class  
};  
class doctor:privateperson  
{  
    //body of class  
}
```



Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

1. Single Inheritance



- If a class is derived from a single class then it is called **single inheritance**.
- Class **B** is derived from class **A**

Example:

```
class Animal
{ public:
    int legs = 4;
};

class Dog : public Animal
{ public:
    int tail = 1;
};
```

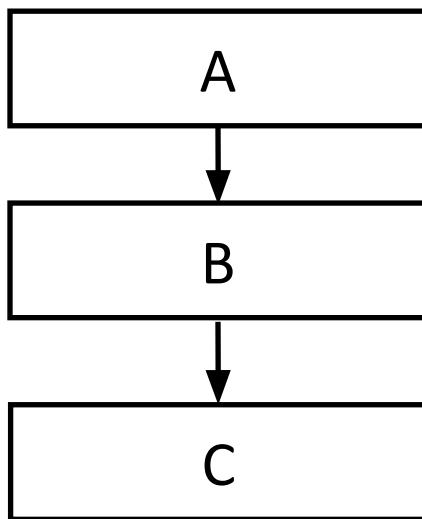
Simple Inheritance Program

```
class Animal{  
    int legs=4;  
public:  
    void display1(){  
        cout<<"\nLegs="<<legs;  
    }  
};  
class Dog : public Animal{  
    bool tail = true;  
public:  
    void display2(){  
        cout<<"\nTail="<<tail;  
    }  
};
```

```
int main()  
{  
    Animal a1;  
    Dog d1;  
    d1.display1();  
    d1.display2();  
}
```

Output:
Legs=4
Tail=1

2. Multilevel Inheritance



- Any class is derived from a class which is derived from another class then it is called **multilevel inheritance**.
- Here, class **C** is derived from class **B** and class **B** is derived from class **A**, so it is called **multilevel inheritance**.

Example:

```
class Person
{
    //content of class person
};

class Student :public Person
{
    //content of Student class
};
```

```
class ITStudent :public
Student
{
    //content of ITStudent
class
};
```

```
class Person{
public:
    void display1(){
        cout<<"\nPerson class";
    }
};

class Student:public Person{
public:
    void display2(){
        cout<<"\nStudent class";
    }
};

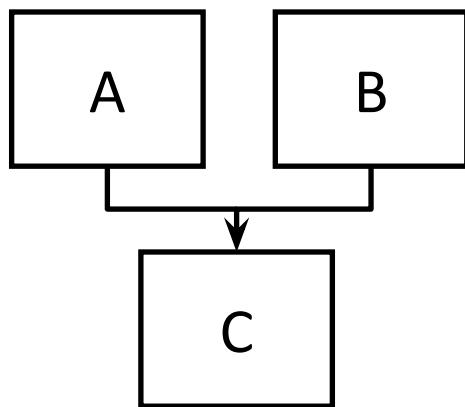
class ITStudent:public Student{
public:
    void display3(){
        cout<<"\nITStudent class";
    }
};
```

```
int main()
{
    Person p;
    Student s;
    ITStudent i;
    p.display1();
    s.display2();
    s.display1();
    i.display3();
    i.display2();
    i.display1();
}
```

Output:

Person class
Student class
Person class
ITStudent class
Student class
Person class

3. Multiple Inheritance



- If a class is derived from more than one class then it is called **multiple inheritance**.
- Here, class **C** is derived from two classes, class **A** and class **B**.

Example:

```
class Liquid
{
    //content of Liquid class
};

class Fuel
{
    //content of Fuel class
};
```

```
class Petrol: public
Liquid, public Fuel
{
    //content of Petrol
class
};
```

```

class Liquid{
public:
    void display1(){
        cout<<"\nLiquid class";
    }
};

class Fuel{
public:
    void display2(){
        cout<<"\nFuel class";
    }
};

class Petrol:public Liquid,public Fuel{
public:
    void display3(){
        cout<<"\nPetrol class";
    }
};

```

```

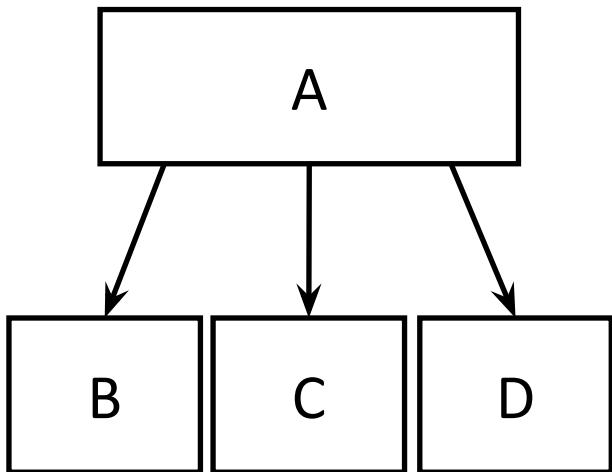
int main()
{
    Liquid l;
    Fuel f;
    Petrol p;
    l.display1();
    f.display2();
    p.display3();
    p.display2();
    p.display1();
}

```

Output:

Liquid class
Fuel class
Petrol class
Fuel class
Liquid class

4. Hierarchical Inheritance



- If one or more classes are derived from one class then it is called **hierarchical inheritance**.
- Here, class **B**, class **C** and class **D** are derived from class **A**.

Example:

```
class Animal
{
    //content of class Animal
};

class Elephant :public Animal
{
    //content of class Elephant
};
```

```
class Horse :public Animal
{
    //content of class Horse
};

class Cow :public Animal
{
    //content of class Cow
};
```

```
class Animal{
public:
void display1(){
    cout<<"\nAnimal Class";
}
};

class Elephant:public Animal{
public:
void display2(){
    cout<<"\nElephant class";
}
};
```

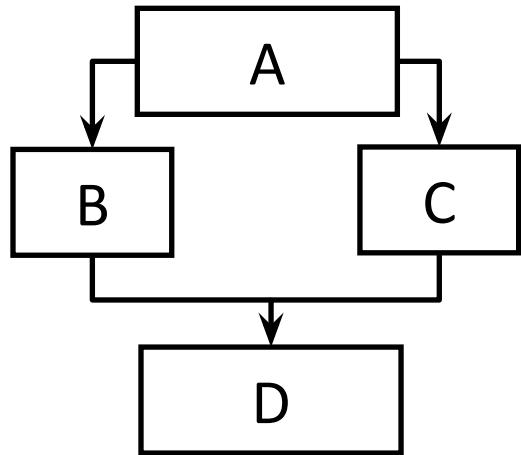
```
int main(){
Animal a; Elephant e; Horse h; Cowc;
a.display1();
e.display2(); e.display1();
h.display3(); h.display1();
c.display4(); c.display1();
}
```

```
class Horse:public Animal{
public:
void display3(){
    cout<<"\nHorse class";
}
};

class Cow:public Animal{
public:
void display4(){
    cout<<"\nCow class";
}
};
```

Output:
Animal Class
Elephant class
Animal Class
Horse class
Animal Class
Cow class
Animal Class

5. Hybrid Inheritance



- It is a combination of any other inheritance types. That is either multiple or multilevel or hierarchical or any other combination.
- Here, class **B** and class **C** are derived from class **A** and class **D** is derived from class **B** and class **C**.
- class **A**, class **B** and class **C** is example of **Hierarchical Inheritance** and class **B**, class **C** and class **D** is example of Multiple Inheritance so this hybrid inheritance is combination of Hierarchical and Multiple Inheritance.

Hybrid Inheritance (Cont...)

```
class Car
{
    //content of class Car
};

class FuelCar:public Car
{
    //content of class FuelCar
};

class ElectricCar:public Car
{
    //content of class ElectricCar
};

class HybridCar:public FuelCar, public ElectricCar
{
    //content of class HybridCar
};
```

```
class Car{
public:
void display1(){
    cout<<"\nCar class";
}
};

class FuelCar:public Car{
public:
void display2(){
    cout<<"\nFuelCar class";
}
};
```

```
int main(){
Car c; FuelCar f; ElecCar e;
HybridCar h;
h.display4();
h.display3();
h.display2();
}
```

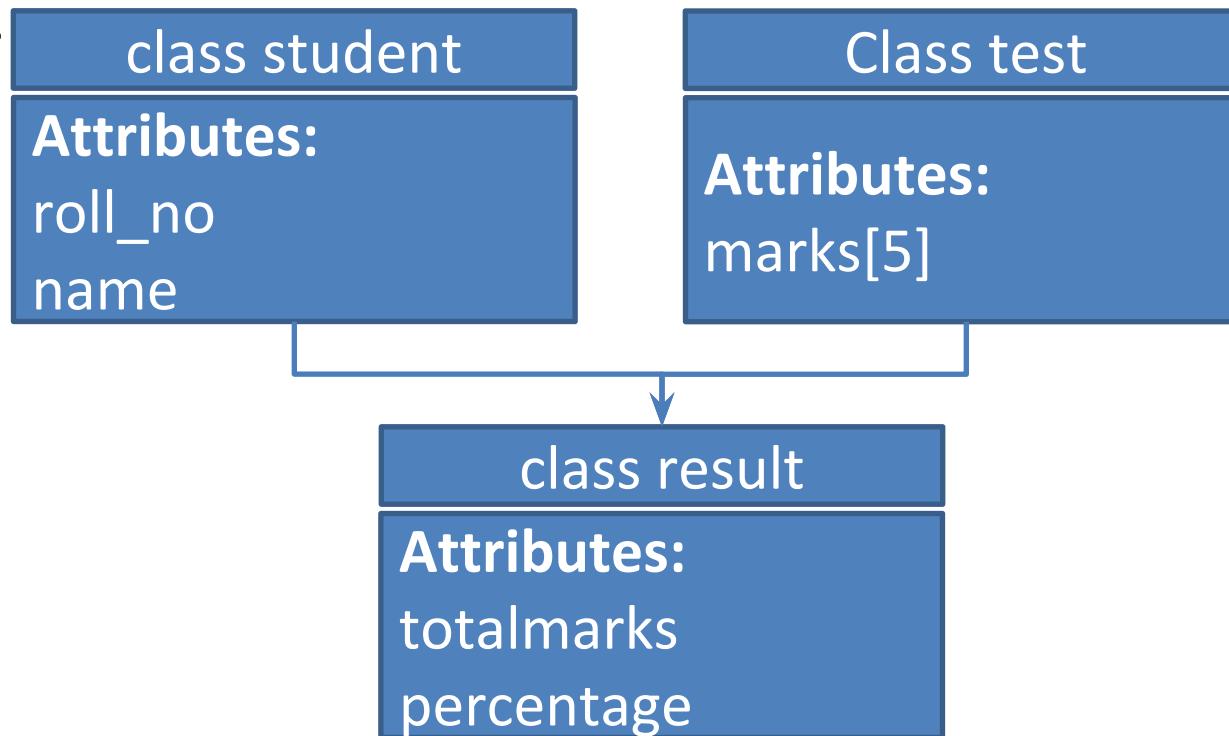
```
class ElecCar:public Car{
public:
void display3(){
    cout<<"\nElecCar class";
}
};

class HybridCar:public
FuelCar, public ElecCar{
public:
void display4(){
    cout<<"\nHybridCar class";
}
};
```

Output:
HybridCar class
ElecCar class
FuelCar class

GTU Program

- Create a class student that stores roll_no, name. Create a class test that stores marks obtained in five subjects. Class result derived from student and test contains the total marks and percentage obtained in test. Input and display information of a student.



Protected access modifier

- **Protected** access modifier plays a key role in inheritance.
- **Protected** members of the class can be accessed within the class and from derived class but cannot be accessed from any other class or program.
- It works like public for derived class and private for other class.

```
class ABC {  
public:  
    void setProtMemb(int i){  
        m_protMemb = i; }  
    void Display(){  
        cout<<m_protMemb<<endl; }  
protected:  
    int m_protMemb;  
    void Protfunc(){  
        cout<<"\nAccess allowed\n"; }  
};
```

```
int main() {  
    ABC a; XYZ x;  
    a.m_protMemb;          //error, m_protMemb is protected  
    a.setProtMemb(0);     //OK, uses public access function  
    a.Display();  
    a.Protfunc();         //error, Protfunc() is protected  
    x.setProtMemb(5);     //OK, uses public access function  
    x.Display();  
    x.useProtfunc();} // OK, uses public access function
```

```
class XYZ : public ABC{  
public:  
    void useProtfunc(){  
        Protfunc(); }  
};
```

```
class A
{
protected:
int a;
public:
void getdata(){
    cout<<"Enter value of a:";
    cin>>a;
}
};

class B:public A
{
public:
void show(){
cout<<"Value of a is:"<<a;
}
};
```

```
int main()
{
B x;
x.getdata();
x.show();
}
```

Class access modifiers

- **public** – Public members are visible to all classes.
- **private** – Private members are visible only to the class to which they belong.
- **protected** – Protected members are visible only to the class to which they belong, and derived classes.

Access modifiers

Base class
members

```
private: x  
protected: y  
public: z
```

public
base class

How inherited base class
members
appear in derived class

```
x is inaccessible  
protected: y  
public: z
```

```
private: x  
protected: y  
public: z
```

private
base class

```
x is inaccessible  
private: y  
private: z
```

```
private: x  
protected: y  
public: z
```

protected
base class

```
x is inaccessible  
protected: y  
protected: z
```

```
class base{
private:
    int z;
public:
    int x;
protected:
    int y;
};

class publicDerived: public base{
    // x is public
    // y is protected
    // z is not accessible from publicDerived
};

class protectedDerived: protected base{
    // x is protected
    // y is protected
    // z is not accessible from
    protectedDerived
};
```

```
class privateDerived: private base
{
    // x is private
    // y is private
    // z is not accessible from
    privateDerived
};
```

Inheritance using Public Access

class Grade

private members:

```
char letter;  
float score;  
void calcGrade();
```

public members:

```
void setScore(float);  
float getScore();  
char getLetter();
```

class Test : public Grade

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);  
void setScore(float);  
float getScore();  
char getLetter();
```

When Test class inherits
from Grade class using →
public class access, it
looks like this:

Inheritance using Private Access

class Grade

private members:

```
char letter;  
float score;  
void calcGrade();
```

public members:

```
void setScore(float);  
float getScore();  
char getLetter();
```

class Test : private Grade

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;  
void setScore(float);  
float getScore();  
float getLetter();
```

public members:

```
Test(int, int);
```

When Test class inherits
from Grade class using →
private class access, it
looks like this:

Inheritance using Protected Access

```
class Grade
```

private members:

```
char letter;  
float score;  
void calcGrade();
```

public members:

```
void setScore(float);  
float getScore();  
char getLetter();
```

```
class Test : protected Grade
```

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

protected members:

```
void setScore(float);  
float getScore();  
float getLetter();
```

When Test class inherits
from Grade class using 
protected class access, it
looks like this:

Visibility of inherited members

Base class visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private			
Protected			
Public			

Method Overriding

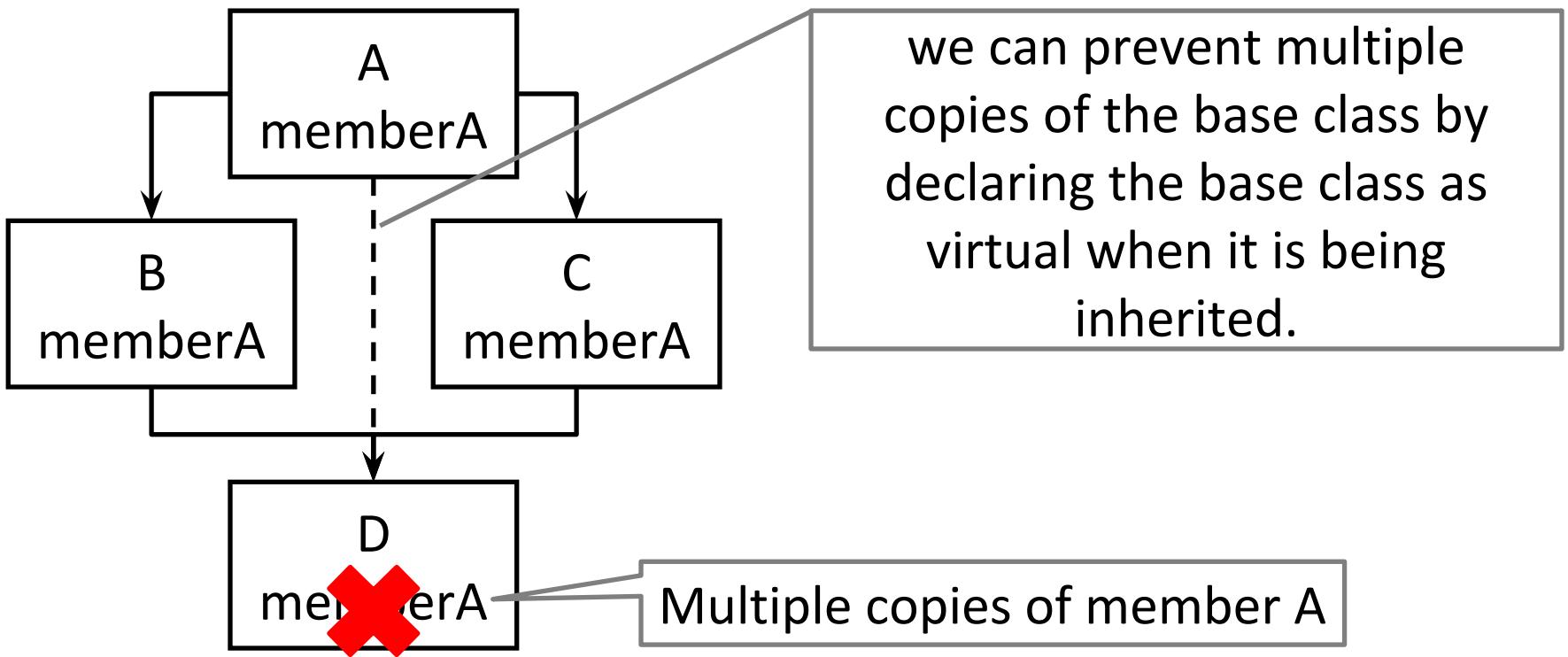
- If base class and derived class have member functions with same name and arguments then method is said to be **overridden** and it is called **function overriding** or **method overriding** in C++.

```
class ABC
{
public:
void display(){
    cout<<"This is parent class";
}
};

class XYZ:public ABC{
public:
void display(){//overrides the display() method of class ABC
    cout<<"\nThis is child class";
}
};

int main(){
XYZ x;
x.display(); //method of class XYZ invokes, instead of class ABC
x.ABC::display();
}
```

Virtual Base Class



Virtual base class (Cont...)

- **Virtual base class** is used to prevent the duplication/ambiguity.
- In hybrid inheritance child class has two direct parents which themselves have a common base class.
- So, the child class inherits the grandparent via two separate paths. It is also called as indirect parent class.
- All the public and protected members of grandparent are inherited twice into child.
- We can stop this duplication by making base class **virtual**.

```
class A
{
public:
    int i;
};

class B:virtual public A
{
public:
    int j;
};

class C: public virtual A
{
public:
    int k;
};
```

```
class D:public B, public C
{
public:
    int sum;
};

int main()
{
    D ob1;
    ob1.i=10;
    ob1.j=20;
    ob1.k=30;
    ob1.sum=ob1.i+ob1.j+ob1.k;
    cout<<ob1.sum;
}
```

Derived class constructor

```
class Base{
    int x;
public:
    Base() { cout << "Base default constructors"; }
};

class Derived : public Base
{ int y;
public:
    Derived() { cout<<"Derived default constructor"; }
    Derived(int i) { cout<<"Derived parameterized constructor"; }
};

int main(){
    Base b;
    Derived d1;
    Derived d2(10);
}
```

Derived class constructor (Cont...)

```
class Base                                int main()
{ int x;
  public:
    Base(int i){
        x = i; cout << "x=" << x;
    }
};

class Derived : public Base {
  int y;
  public:
    Derived(int i,int j) : Base(j)
    { y = i; cout << "y=" << y;
    }
};

int main()
{
    Derived d(10,20) ;
}
```

Execution of base class constructor

Method of inheritance	Order of execution
class Derived: public Base { };	Base(); Derived();
class C: public A, public B { };	A();//base(first) B();//base(Second) C();derived
class C:public A, virtual public B { };	B();//virtual base A();//base C();derived

Unit-5

Pointers, Virtual functions and polymorphism



Polymorphism

- Pointers in C++
- Pointers and Objects
- this pointer
- virtual and pure virtual functions
- Implementing polymorphism

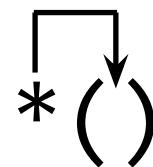
Pointer

- Pointer is a variable that holds a memory address, of another variable.

```
int a = 25;  
int *p;  
p = &a;
```



cout<<"&a:"<<&a;	&a:1000
cout<<"p:"<<p;	p:1000
cout<<"&p:"<<&p;	&p:2000
cout<<"*p:"<<*p;	*p:25
cout<<"*(&a):"<<*(&a);	*(&a):25
(*p)++;	
cout<<"*p:"<<*p;	*p:26
cout<<"a:"<<a;	a:26

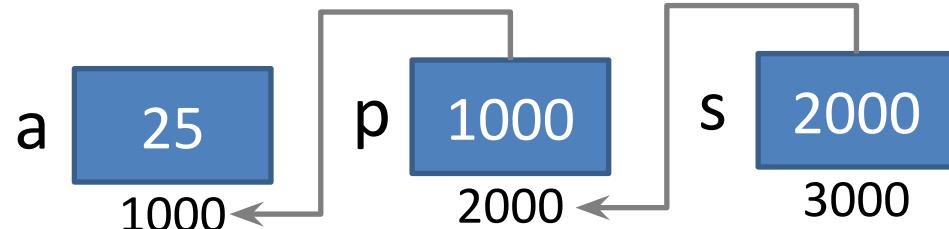


* Indicates value
at address

Pointer (Cont...)

- Pointer to pointer is a variable that holds a memory address, of another pointer variable.

```
int a = 25;  
int *p, **s;  
p = &a;  
s = &p;
```



```
cout<<"\n*p:"<<*p; *p:25
```

```
cout<<"\n*s:"<<*s; *s:1000
```

```
cout<<"\n**s:"<<**s; **s:25
```

```
cout<<"\n*(\*(\&p)):"<<*(\*(\&p)); *(\*(\&p)):25
```

```
cout<<"\n***(\&s):"<<***(\&s); ***(\&s):25
```

Pointer to arrays

```
int main ()
{
    int arr[5] = {10,20,30,40,50};
    int *ptr;
    ptr = &arr[0];
    for ( int i = 0; i < 5; i++ )
    {
        cout <<*(ptr+i) << i << ":";
        cout <<*(ptr + i) << endl;
    }
    return 0;
}
```

Also, written as
ptr = arr;

0	10	1000	ptr
1	20	1002	
2	30	1004	
3	40	1006	
4	50	1008	

Output:

*(ptr + 0) : 10
*(ptr + 1) : 20
*(ptr + 2) : 30
*(ptr + 3) : 40
*(ptr + 4) : 50

Pointers and objects

- Just like pointers to normal variables and functions, we can have pointers to class members (variables and methods).

```
class ABC
{
    public:
        int a=50;
};

int main()
{
    ABC ob1;
    ABC *ptr;
    ptr = &ob1;
    cout << ob1.a;
    cout << ptr->a; // Accessing member with pointer
}
```

When accessing members of a class given a pointer to an object, use the arrow (\rightarrow) operator instead of the dot operator.

Pointers and objects (Cont...)

```
class demo{
    int i;
public:
    demo(int x){
        i=x; }
    int getdata(){
        return i;}
};

int main()
{
    demo d(55),*ptr;
    ptr=&d;
    cout<<ptr->getdata();
}
```

Pointers and objects (Cont...)

```
class demo{  
    int i;  
public:  
    demo(int x){  
        i=x; }  
    int getdata(){  
        return i; }  
};  
int main()  
{  
    demo d[3]={55,66,77};  
    demo *ptr=d; //similar to *ptr=&d[0]  
    for(int i=0;i<3;i++)  
    {  
        cout<<ptr->getdata()<<endl;  
        ptr++;  
    }  
}
```

- When a pointer incremented it points to next element of its type.
- An integer pointer will point to the next integer.
- The same is true for pointer to objects

Program

- Create a class student having private members marks, name and public members rollno, getdata(), printdata(). Demonstrate concept of pointer to class members for array of 3 objects.

this pointer

```
class Test
{
    int mark;
    float spi;
public:
    void SetData(){
        this->mark=70;
        this->spi=6.55;
    }
    void DisplayData(){
        cout << "Mark= "<<mark;
        cout << "spi= "<<spi;
    }
};
```

int main()

```
{
```

Test o1;

```
o1.SetData();
o1.DisplayData();
```

```
}
```

- Within member function, the members can be accessed directly, without any object or class qualification.
- But implicitly members are being accessed using **this** pointer

- When a member function is called, it automatically passes a **pointer** to invoking object.

this pointer(Cont...)

- ‘this’ pointer represent an object that invoke or call a member function.
- It will point to the object for which member function is called.
- It is automatically passed to a member function when it is called.
- It is also called as implicit argument to all member function.

Note:

- ✓ Friend functions can not be accessed using **this** pointer, because friends are not members of a class.
- ✓ Only member functions have a **this** pointer.
- ✓ A **static** member function does not have **this** pointer.

this pointer (Cont...)

```
class sample
{
    int a,b;
public:
    void input(int a,int b){
        this->a = a + b;
        this->b = a - b;
    }
    void output(){
        cout<<"a = "<<a;
        cout<<"b = "<<b;
    }
};

int main()
{
    sample ob1;
    int a=5,b=8;
    ob1.input(a,b);
    ob1.output();
}
```

this pointer is used when local variable's name is same as member's name.

```
class person
{
    int age;
public:
    person(int x){age=x;}
    void display(){cout<<"Age="<<age;}
    person& olderperson(person p){
        if (age > p.age)
            return *this;
        else
            return p;
    }
};

int main()
{
    person r(35),h(30);
    person o=r.olderperson(h);
    o.display();
}
```

this pointer is used to return reference to the calling object

this pointer (Cont...)

```
class Test
{
    int x; int y;
public:
    Test& setX(int a) { x = a; return *this; }
    Test& setY(int b) { y = b; return *this; }
    void print() {
        cout << "x = " << x ;
        cout << " y = " << y;
    }
};
```

```
int main()
{
    Test obj1;
    obj1.setX(10).setY(20);
    obj1.print();
}
```

this pointer is used to return reference to the calling object

Pointer to derived class

- We can use pointers not only to the base objects but also to the objects of derived classes.
- A single pointer variable of base type can be made to point to objects belonging to base as well as derived classes.

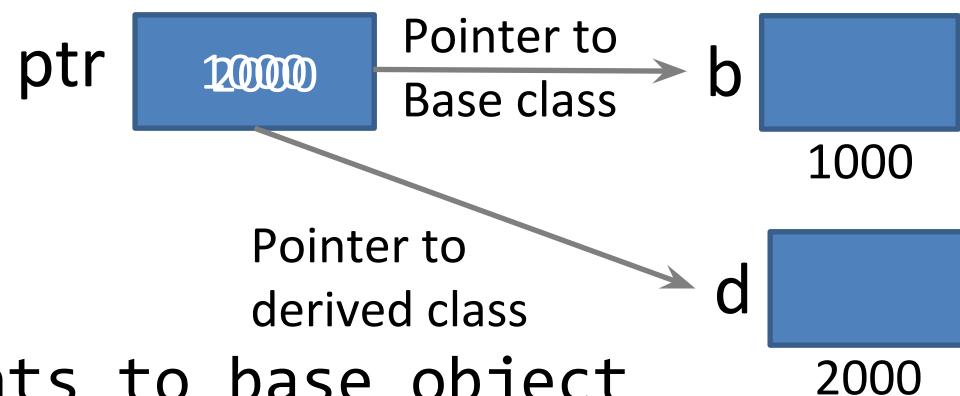
For example:

```
Base *ptr;
```

```
Base b;
```

```
Derived d;
```

```
ptr = &b; //points to base object
```



- We can make `ptr` to point to the object `d` as follows

```
ptr = &d; //base pointer point to derived object
```

```
class Base {  
public:  
void show(){  
    cout << "Base\n"; }  
};  
class Derv1 : public Base {  
public:  
void show(){  
    cout << "Derv1\n"; }  
};  
int main(){  
Derv1 dv1;  
Base* ptr;  
ptr = &dv1;  
ptr->show();  
((Derv1 *)ptr)->show();  
}
```

Derived type casted to
base type
Explicitly
casted into derived type

Output:
Base
Derv1

Pointer to derived class (Cont...)

- We can access those members of derived class which are **inherited from base class** by base class pointer.
- But we cannot access original member of derived class which are **not inherited** from base class using base class pointer.
- We can access original member of derived class which are not inherited by using pointer of derived class.

Program

```
class base
{
public:
int b;
void show()
{
    cout<<"\nb="<<b;
}
};

class derived : public base
{
public:
int d;
void show()
{
    cout<<"\n b="\b<<b<<"\n d="\d;
}
};
```

Program (Cont...)

```
int main(){
base B1;
derived D1;
base *bptr;
bptr=&B1;
cout<<"\nBase class pointer assign address of base class object";
bptr->b=100;
bptr->show();
bptr=&D1;
bptr->b=200;
cout<<"\nBase class pointer assign address of derived class
object";
bptr->show();
derived *dptr;
dptr=&D1;
cout<<"\nDerived class pointer assign address of derived class
object";
dptr->d=300;
dptr->show();
}
```

Program (Cont...)

```
int main(){
base B1;
derived D1;
base *bptr;
bptr=&B1;
cout<<"\nBase class pointer assign address of base class object";
bptr->b=100;
bptr->show();
bptr=&D1;
bptr->b=200;
cout<<"\nBase class pointer assign address of derived class
object";
bptr->show();
derived *dptr;
dptr=&D1;
cout<<"\nDerived class pointer assign address of derived class
object";
dptr->d=300;
dptr->show();
}
```

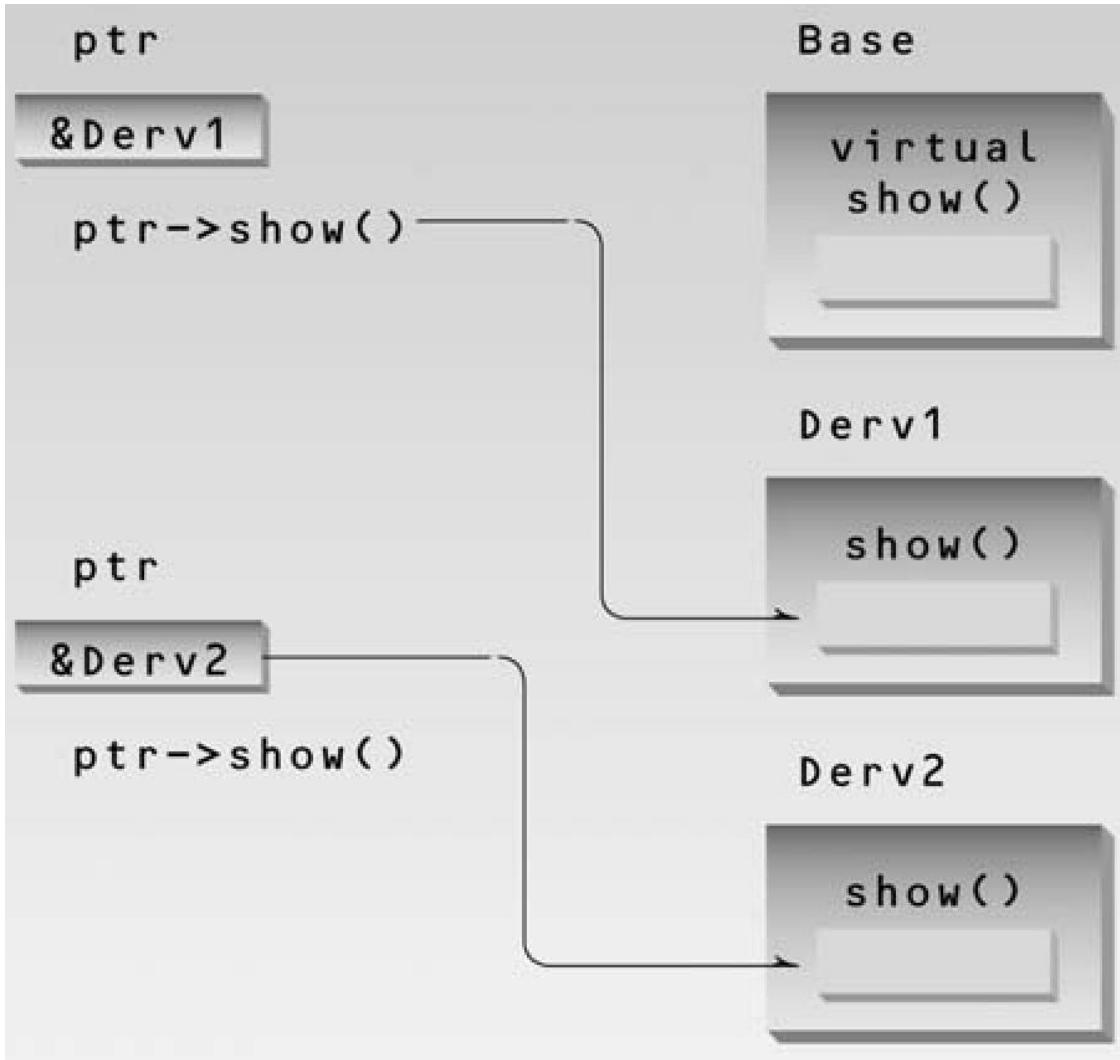
Virtual Base Function

- It is a run time polymorphism.
- Base class and derived class have same function name and base class pointer is assigned address of derived class object then also pointer will execute base class function.
- After making virtual function, the compiler will determine which function to execute at run time on the basis of assigned address to pointer of base class.

```
class Base {  
public:  
virtual void show(){  
    cout << "Base\n"; }  
};  
class Derv1 : public Base {  
public:  
void show(){  
    cout << "Derv1\n"; }  
};  
class Derv2 : public Base {  
public:  
void show(){  
    cout << "Derv2\n"; }  
};
```

```
int main()  
{  
    Derv1 dv1;  
    Derv2 dv2;  
    Base* ptr;  
    ptr = &dv1;  
    ptr->show();  
    ptr = &dv2;  
    ptr->show();  
}
```

Output:
Derv1
Derv2



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

Rules for virtual base function

1. The virtual functions must be member of any class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. We cannot have virtual constructors, but we can have virtual destructors.
7. The derived class pointer cannot point to the object of base class.
8. When a base pointer points to a derived class, then also it is incremented or decremented only relative to its base type. Therefore we should not use this method to move the pointer to the next object.
9. If a virtual function is defined in base class, it need not be necessarily redefined in the derived class. In such cases, call will invoke the base class.

Pure virtual functions

- A **pure virtual function** means ‘do nothing’ function.
- We can say empty function. A **pure virtual function** has no definition relative to the base class.
- Programmers have to redefine **pure virtual function** in derived class, because it has no definition in base class.
- A class containing **pure virtual function** cannot be used to create any direct objects of its own.
- This type of class is also called as **abstract class**.

Syntax:

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

OR

```
virtual void display() {}
```

Program

```
class Shape{  
protected:  
    float x;  
public:  
    void getData(){cin >> x;}  
    virtual float calculateArea() = 0;  
};  
class Square : public Shape  
{  
public:  
    float calculateArea()  
    { return l*l; }  
};  
class Circle : public Shape  
{  
public:  
    float calculateArea()  
    { return 3.14*l*l; }  
};
```

This is called abstract class

Program

```
int main()
{
    Square s;
    Circle c;
    cout << "Enter length to calculate the area of a square:";
    s.getData();
    cout<<"Area of square: " << s.calculateArea();
    cout<<"Enter radius to calculate the area of a circle: ";
    c.getData();
    cout << "Area of circle: " << c.calculateArea();

}
```

Output:

Enter length to calculate the area of a square: 10

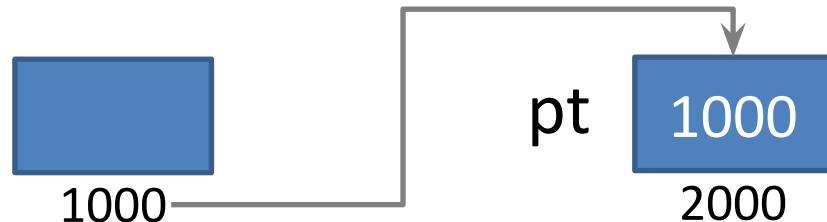
Area of square: 100

Enter radius to calculate the area of a circle: 9

Area of circle: 254.34

Memory allocation using `new` operator

- `new` is used to dynamically allocate memory
- `new` finds a block of the correct size and returns the address of the block.
- Assign this address to a pointer.

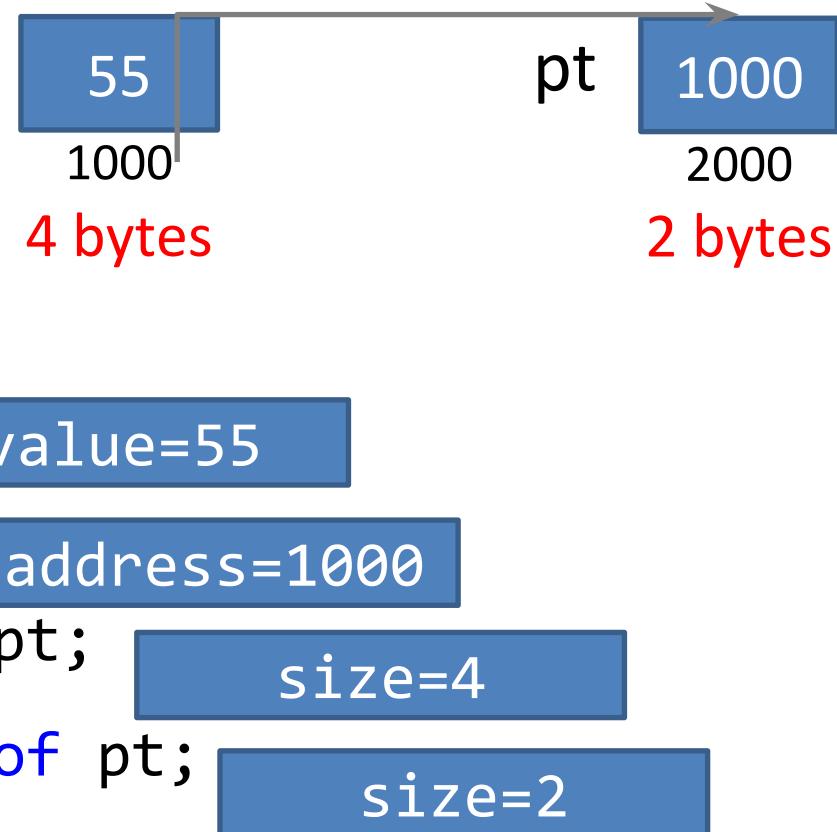


```
int *pt = new int;
```

- `new int` part tells the program you want some `new` storage of size `int`.
- Then it finds the memory and returns the address.
- Next, assign the address to `*pt`.
- Now `pt` is the address and `*pt` is the value stored there.

Program

```
int main ()  
{  
    float *pt = new float;  
    *pt = 55;  
  
    cout<<"value="<< *pt;  
    cout<<"\naddress="<< pt;  
    cout<<"\nsize="<< sizeof *pt;  
    cout<<"\nsize ptr="<< sizeof pt;  
}
```



Free memory using `delete` operator

- `delete` operator frees memory allocated using `new`.

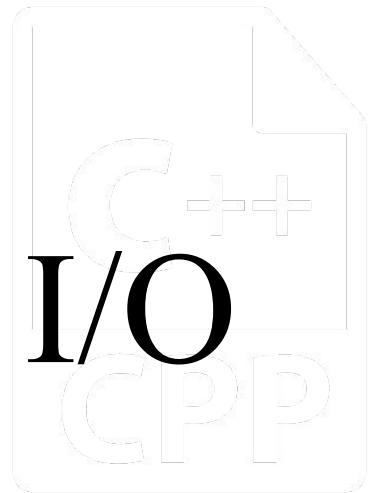
```
int * ps = new int; // allocate memory with new  
. . . // use the memory  
delete ps; // free memory with delete when done
```

- it doesn't remove the pointer `ps` itself. You can reuse `ps`, to point to another new allocation.

```
int * ps = new int; // ok  
delete ps; // ok  
delete ps; // not ok now  
int jugs = 5; // ok  
int * pi = &jugs; // ok  
delete pi; //not allowed, memory not allocated by new
```

Unit-6

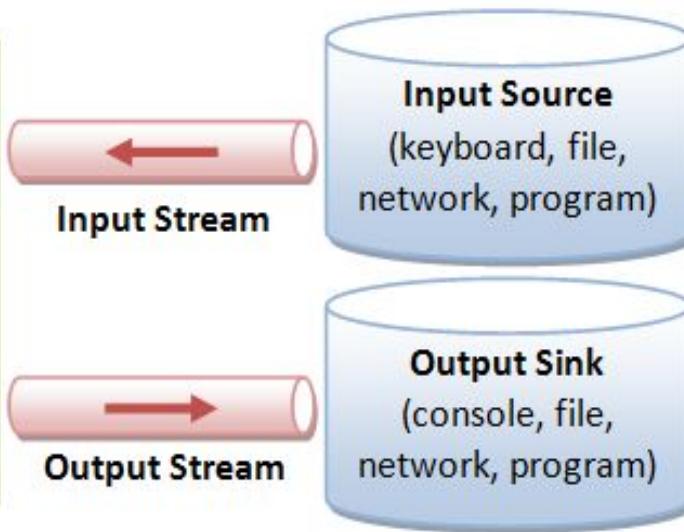
Managing Console I/O Operations



I/O and File Management

- Concept of streams
- cin and cout objects
- C++ stream classes
- Unformatted and formatted I/O
- Manipulators
- File stream
- C++ File stream classes
- File management functions
- File modes
- Binary and random Files

Concept of streams



Internal Data Formats:

- Text: char, wchar_t
- int, float, double, etc.

External Data Formats:

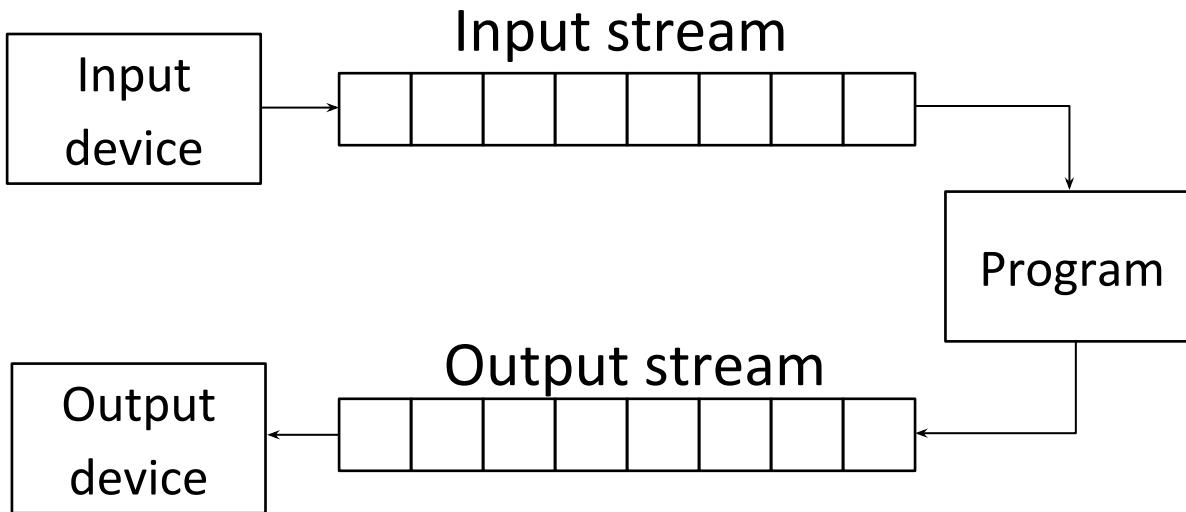
- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

- A **stream** is a general name given to a flow of data.
- A **stream** is a sequence of bytes.
- The source stream that provides data to programs is called **input stream**.
- The destination stream receives output from the program is called **output stream**.

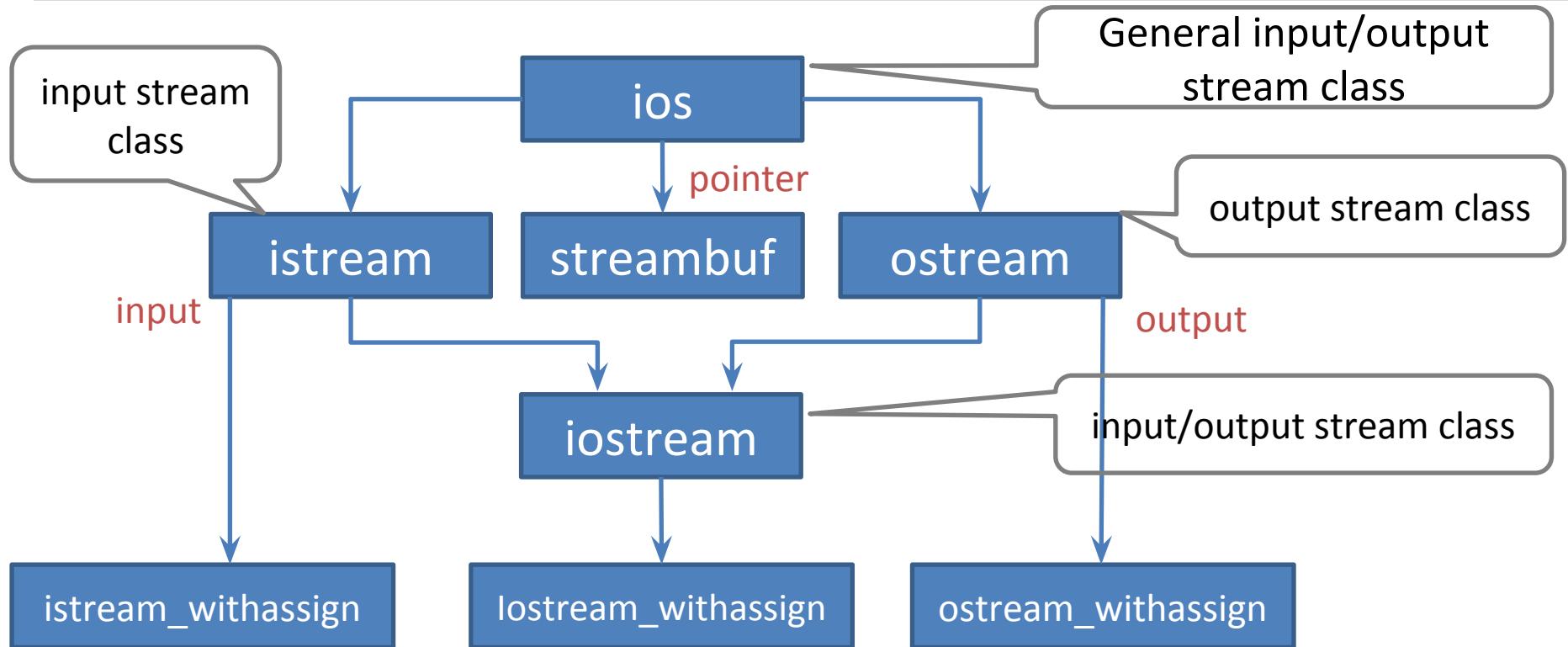
Concept of streams(Cont...)

- In C++ a stream is represented by an object of a particular class.
- So far we've used the **cin** and **cout** stream objects.
- Different streams are used to represent different kinds of data flow.
- The classes used for input/output to the **devices** are declared in the **IOSSTREAM** file.
- The classes used for **disk file** are declared in the **FSTREAM** file.

Data input output streams



Stream class for console I/O operations



- **istream_withassign, ostream_withassign, and iostream_withassign**
 - These three classes implement appropriate functionality based on their base class **ios**.
 - They add assignment operators to its base classes.
 - They implement **operator<<**, **operator>>**, **get**, **put**, **read**, **write**, **flush** methods of **ostream**.
 - **cout**, which is directed to video display, is predefined object of **iostream** class.
 - **cout** is also **separated** to a header file **fstream.h**.
 - **cout** is a **class** object of **ostream** class that only one copy of **ostream** withassign.
 - **cin** is an object of **istream** withassign.

put(), get(), getline(), write() - Unformatted I/O Operations

```
char ch;  
cin.get(ch);  
ch=cin.get();  
cin>>ch;  
cout.put(ch);  
cout.put('x');
```

Get a character from keyboard

Similar to `cin.get(c);`

The operator `>>` can also be used to read a character but it will skip the white spaces and newline character.

`put()` function can be used to display value of variable `ch` or character.

```
char name[20];  
cin.getline(name, 10);  
cin>>name;  
cout.write(name, 10);
```

`getline()` reads whole line of text that ends with newline character or upto $(size-1)$.

`cin` can read strings that do not contain white

`write()` displays string of given size, if the size is greater than the length of line, then it displays the bounds of line.

ios Format Functions

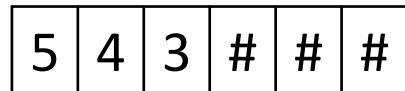
Function	Task
width()	To specify the required field size for displaying an output value
precision()	To specify number of digits to be displayed after the decimal point of a float value.
fill()	To specify a character that is used to fill the unused portion of a field.
setf()	To specify format flags that can control the form of output.
unsetf()	To clear the flags specified

Example:

output:

```
cout.setf(ios::right,ios::adjustfield);
cout.width(6);
cout.fill('#');
cout<<"543";
```

output:



Flags and bit fields

Format required	Flag (arg1)	Bit-field (arg2)
Left justified output	ios::left	ios::adjustfield
Right justified output	ios::right	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal base	ios::dec	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

`setf(arg1, arg2)`

arg-1: one of the formating flags.

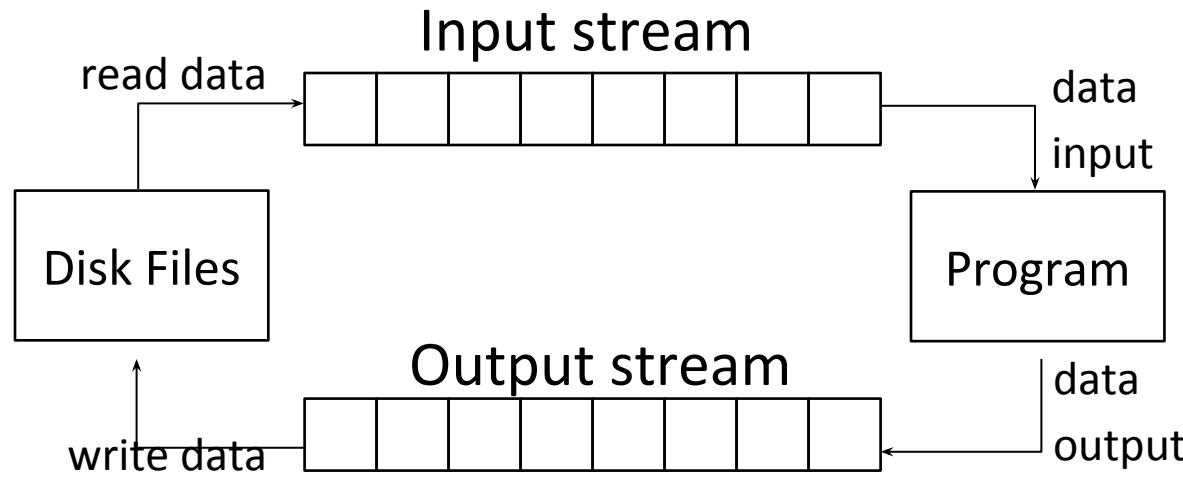
arg-2: bit field specifies the group to which the formatting flag belongs.

Manipulators for formatted I/O operations

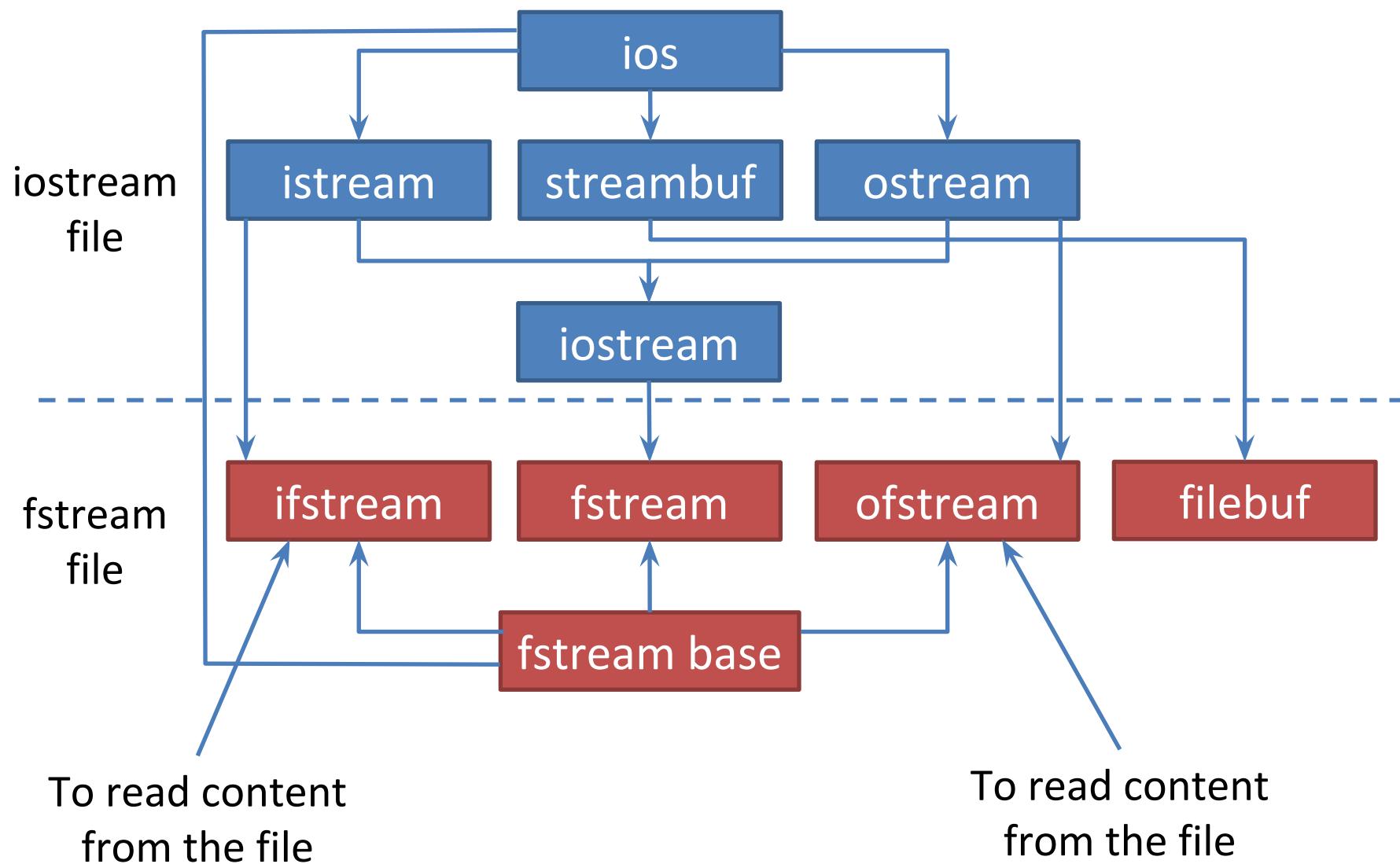
- **Manipulators** are special functions that can be included in the I/O statements to alter the format parameters of a stream.
- To access manipulators, the file **<iomanip>** should be included in the program.

Function	Manipulator	Meaning
width()	setw()	Set the field width.
precision()	setprecision()	Set the floating point precision.
fill()	setfill()	Set the fill character.
setf()	setiosflags()	Set the format flag.
unsetf()	resetiosflags()	Clear the flag specified.
“\n”	endl	Insert a new line and flush stream.

File input output streams



File stream classes for file operations



File stream classes

class	contents
fstreambase	<ul style="list-style-type: none">▪ Provides operations common to the file streams.▪ Contains open() and close() functions.
ifstream	<ul style="list-style-type: none">▪ Provides input operations.▪ Contains open() with default input mode.▪ Inherits get(), getline(), read(), seekg() and tellg() functions from istream.
ofstream	<ul style="list-style-type: none">▪ Provides output operations.▪ Contains open() with default output mode.▪ Inherits put(), seekp(), tellp() and write() functions from ostream.
fstream	<ul style="list-style-type: none">▪ Provides support for simultaneous input and output operations.▪ Inherits all the functions from istream and ostream from iostream.
filebuf	<ul style="list-style-type: none">▪ Its purpose is to set the file buffers to read and write.

File handling steps

1. Open / Create a file
2. Read / Write a file
3. Close file

Opening a file

```
ofstream outFile("sample.txt"); //output only  
ifstream inFile("sample.txt"); //input only
```

```
ofstream outFile;  
outFile.open("sample.txt");
```

This creates **outFile** as an **ofstream** object that manages the output

```
ifstream inFile;  
inFile.open("sample.txt");
```

This object can be any valid C++ name such as myfile, o_file .

- Syntax file **open()** function:

```
stream-object.open("filename", mode);
```

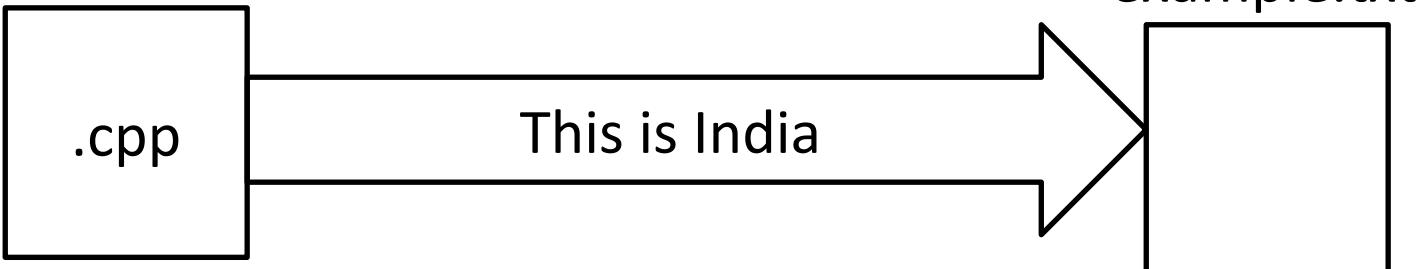
- By default **ofstream** opens file for writing only and **ifstream** opens file for reading only.

File opening modes

Parameter	Meaning
ios :: app	Append to end-of-file
ios :: ate	Go to end-of-file on opening
ios :: binary	Binary file
ios :: in	Open file for reading only
ios :: out	Open file for writing only
ios :: trunc	Delete content of file if exists
ios :: nocreate	Open fails if the file does not exists
ios :: noreplace	Open fails if the file already exists

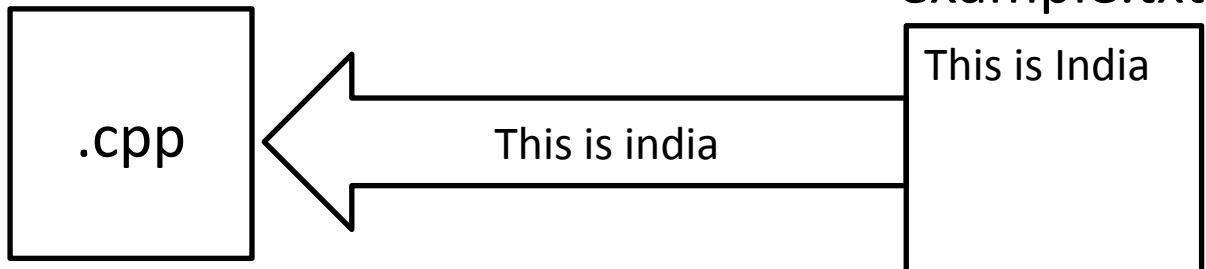
File operations

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
    ofstream myfile;
    myfile.open("example.txt",ios::out);
    myfile << "This is India.\n";
    myfile << "It is a great country.\n";
    myfile.close();
}
```



File operations (Cont..)

```
int main ()  
{  
    char line[50];  
    ifstream rfile;  
    rfile.open("example.txt",ios::in)  
    rfile.getline(line,50);  
    //we can also write rfile>>line;  
  
    cout<<line;  
    rfile.close();  
}
```



```
int main()
```

```
{
```

```
    char product[20];
```

```
    int price;
```

```
    cout<<"Enter product name=";
```

```
    cin>>product;
```

```
    cout<<"Enter price=";
```

```
    cin>>price;
```

```
    ofstream outfile("stock.txt");
```

```
    outfile<<product<<endl;
```

```
    outfile<<price;
```

```
    ifstream infile("stock.txt");
```

```
    infile>>product;
```

```
    infile>>price;
```

```
    cout<<product<<endl;
```

```
    cout<<price;
```

File operations program

Opening a file to write
data into file

Opening a file to read
data from file

File pointers

- Each file has two associated pointers known as the **file pointers**.
- One of them is called **input pointer (or get pointer)** and the other is called **output pointer (or put pointer)**.
- **Input pointer** is used for reading the content of a given file location.
- **Output pointer** is used for writing to a given file location.

Functions for manipulation of file pointers

Function	Meaning
seekg()	Moves get pointer (input) to specified location
seekp()	Moves put pointer (output) to specified location
tellg()	Gives current position of the get pointer
tellp()	Gives current position of the put pointer

```
ifstream fin;  
ofstream fout;
```

```
fin.seekg(30); //move the get pointer to byte number 30 in the file  
fout.seekp(30); //move the put pointer to byte number 30 in the file  
int posn = fin.tellg();  
int posn = fout.tellp();
```

Functions for manipulation of file pointers

Another prototype

```
seekg ( offset, direction );  
seekp ( offset, direction );
```

Function	Meaning
ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position of the stream pointer
ios::end	offset counted from the end of the stream

GTU Program

- Write a program that opens two text files for reading data. It creates a third file that contains the text of first file and then that of second file (text of second file to be appended after text of the first file, to produce the third file).

```
int main() {
    fstream file1,file2,file3;
    file1.open("one.txt",ios::in);
    file2.open("two.txt",ios::in);
    file3.open("three.txt",ios::app);
    char ch1,ch2;
    while(!file1.eof())
    {
        file1.get(ch1); cout<<ch2<<endl;
        file3.put(ch1);
    }
    file1.close();
    while(!file2.eof())
    {
        file2.get(ch2); cout<<ch1<<endl;
        file3.put(ch2);
    }
    file2.close(); file3.close();
}
```

write() and read() functions

- The functions **write()** and **read()**, different from the functions **put()** and **get()**, handle the data in binary form.

```
infile.read ((char * ) &V,sizeof(V));
```

```
outfile.write ((char * ) &V ,sizeof(V));
```

- These functions take two arguments. The first is the address of the variable V, and the second is the length of that variable in bytes.
- The address of the variable must be cast to type char*(i.e pointer to character type).

Reading & Writing class objects

```
class inventory
{
    char name[10];
    float cost;
public:
void readdata()
{
    cout<<"Enter Name=";
    cin>>name;
    cout<<"Enter cost=";
    cin>>cost;
}
void displaydata()
{
    cout<<"Name="<<name<<endl;
    cout<<"Cost="<<cost;
}
};
```

Reading & Writing class objects

```
int main()
{
    inventory ob1;
    cout<<"Enter details of product\n";

    fstream file;
    file.open("stock.txt",ios::in | ios::app);

    ob1.readdata();
    file.write((char *)&ob1,sizeof(ob1));

    file.seekg(0);

    file.read((char *)&ob1,sizeof(ob1));

    ob1.displaydata();
    file.close();
}
```



DESIGN

UML Class Diagrams

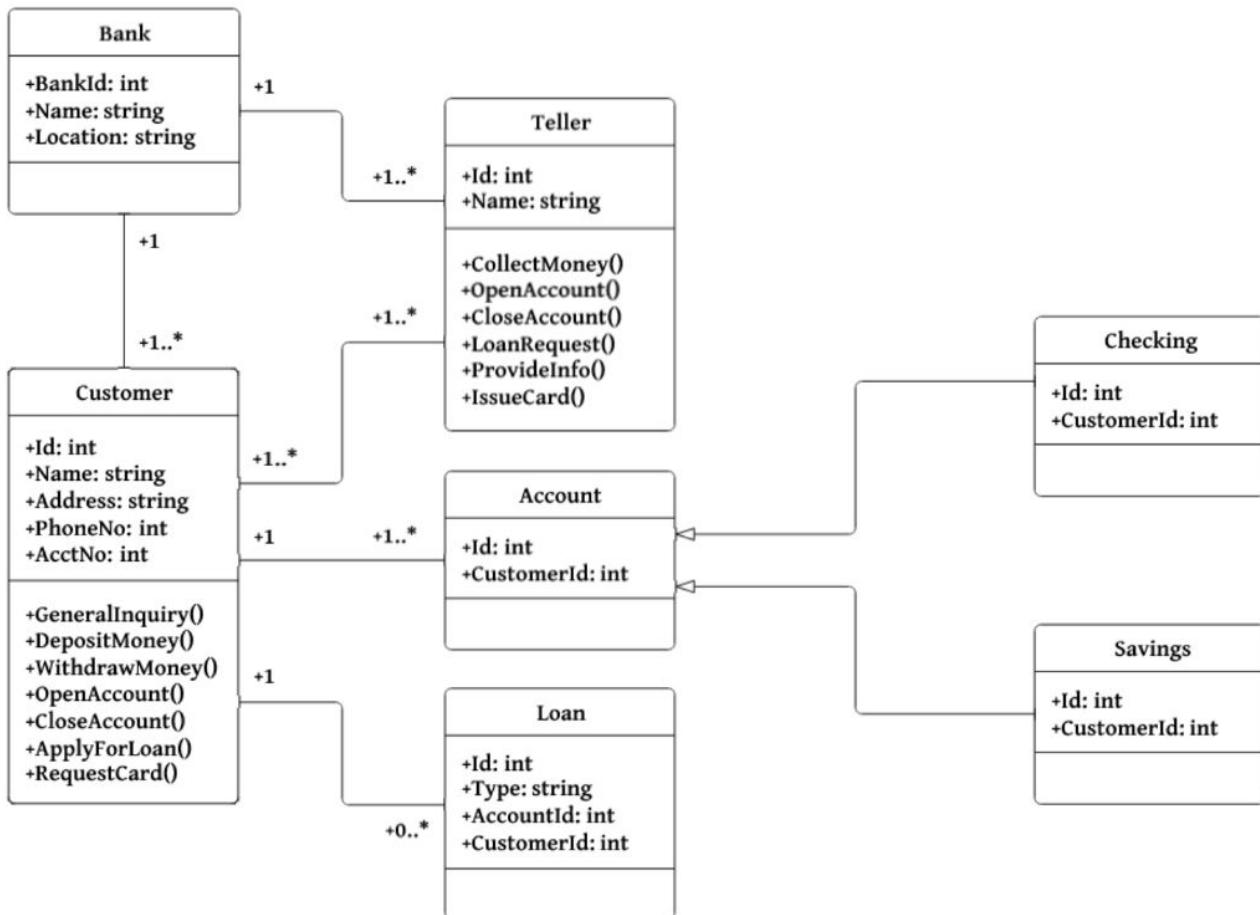
UML Class Diagrams Tutorial, Step by Step

- This is a short tutorial on UML Class Diagrams.
- We'll discuss what they are, why they're needed, some technical stuff, and then we'll dive into an example.

What is a Class Diagram?

- Suppose you have to design a system. Before implementing a bunch of classes, you'll want to have a conceptual understanding of the system
 - What classes do I need?
 - What functionality and information will these classes have?
 - How do they interact with one another?
 - Who can see these classes?
- That's where class diagrams come in. Class diagrams are a neat way of visualizing the classes in your system *before* you actually start coding them up. They're a static representation of your system structure.

Example of a Class Diagram for a Banking System



Why Do We Need Class Diagrams?

- This is a fairly simple diagram.
- However, as your system scales and grows, it becomes increasingly difficult to keep track of all these relationships.
- Having a precise, organized, and straight-forward diagram to do that for you is integral to the success of your system.

DESIGN

- Planning and modeling ahead of time makes programming much easier.
- Besides that, making changes to class diagrams is easy, whereas coding different functionality after the fact is kind of annoying.
- When someone wants to build a house, they don't just grab a hammer and get to work. They need to have a blueprint — a design plan — so they can ANALYZE & modify their system.
- You don't need much technical/language-specific knowledge to understand it.

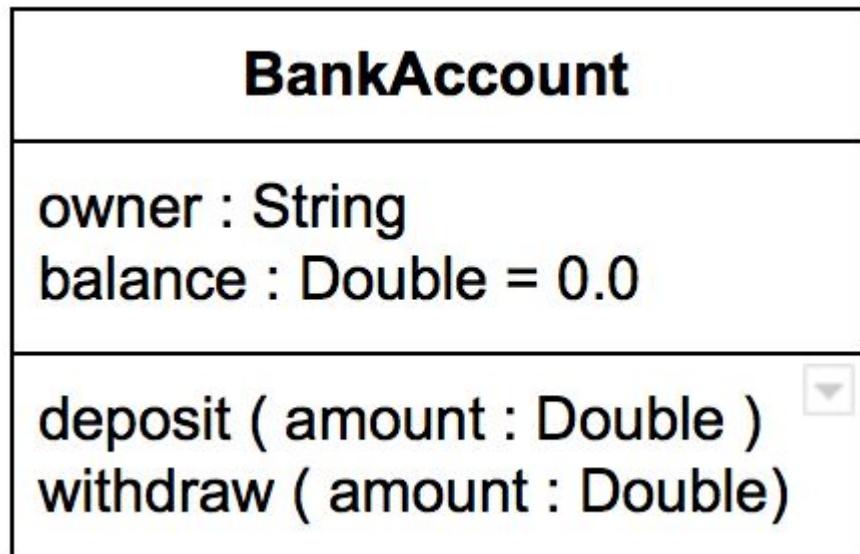
The background features a dynamic, abstract design with flowing, translucent shapes in shades of yellow, green, blue, and red. A solid blue horizontal bar spans across the top right, containing the word "DESIGN" in large, bold, yellow capital letters.

DESIGN

SOME TECHNICAL STUFF

Class Representation in UML

- A class is represented as a box with 3 compartments.
 - The uppermost one contains the class name.
 - The middle one contains the class attributes
 - The last one contains the class methods.



They Adhere to a Convention:

- attribute name : type
- method name (parameter: type)
- if you'd like to set a default value to an attribute do as above
balance : Dollars = 0
- if a method doesn't take any parameters then leave the parentheses empty. Ex: checkBalance()

Visibility of Class Members

- Class members (attributes and methods) have a specific visibility assigned to them. See table below for how to represent them in UML.

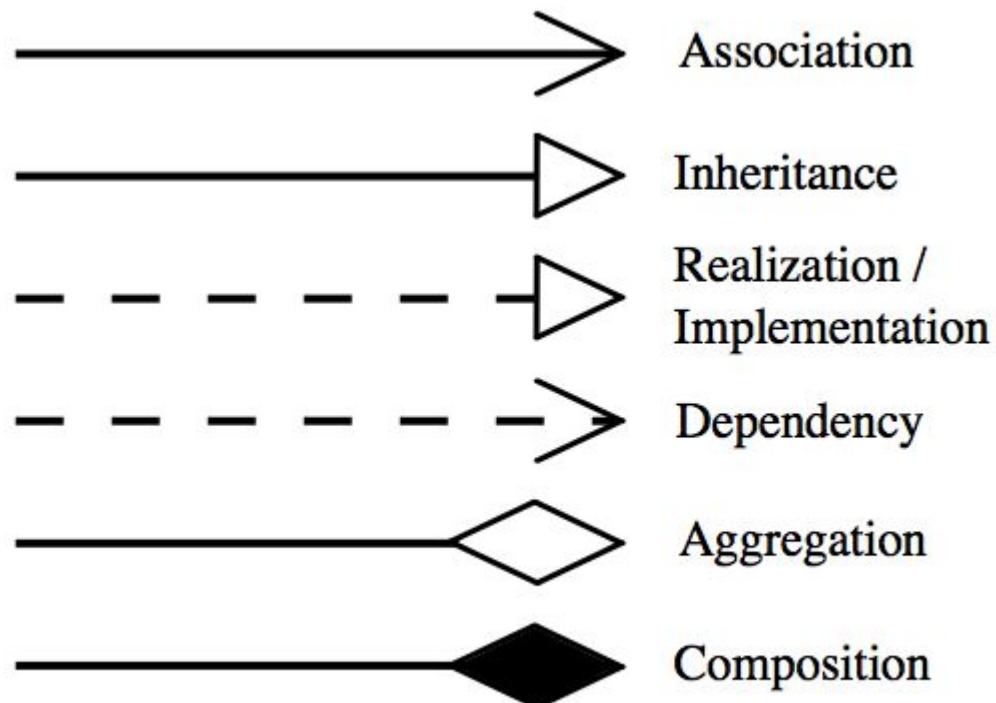
public	+	anywhere in the program and may be called by any object within the system
private	-	the class that defines it
protected	#	(a) the class that defines it or (b) a subclass of that class
package	~	instances of other classes within the same package

Visibility of Members in the BankAccount class

- We made the `owner` and balance private as well as the withdraw method.
- We kept the deposit method public. (Anyone can put money in, but not everyone can take money out. Just as we like it.)

BankAccount
-owner : String -balance : Double = 0.0
+deposit (amount : Double) -withdraw (amount : Double)

Relationships

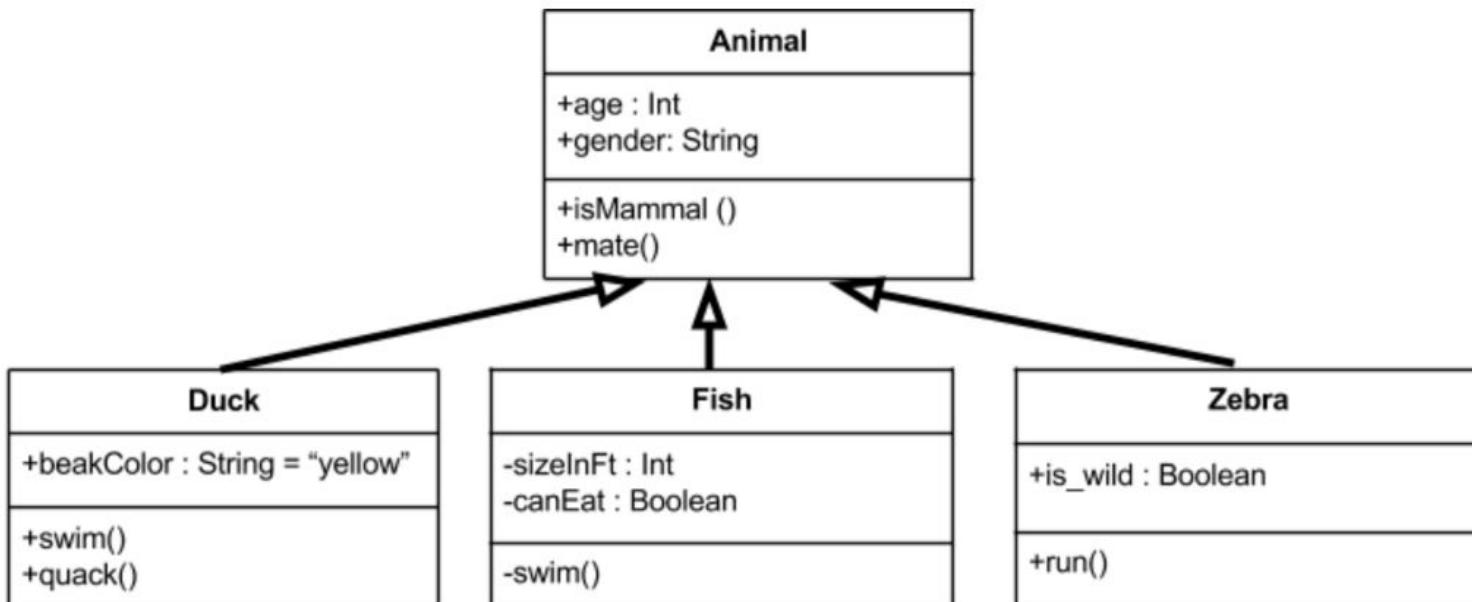


Association

- An association is a relationship between two separate classes. It joins two entirely separate entities.
- There are four different types of association:
 - Bi-directional
 - Uni-directional
 - Aggregation (includes composition aggregation)
 - Reflexive.
- Bi-directional and uni-directional associations are the most common ones.
- This can be specified using multiplicity (one to one, one to many, many to many, etc.).
- A typical implementation in Java is through the use of an instance field. The relationship can be bi-directional with each class holding a reference to the other.

Inheritance

- Indicates that child (subclass) is considered to be a specialized form of the parent (super class).
- For example consider the following:

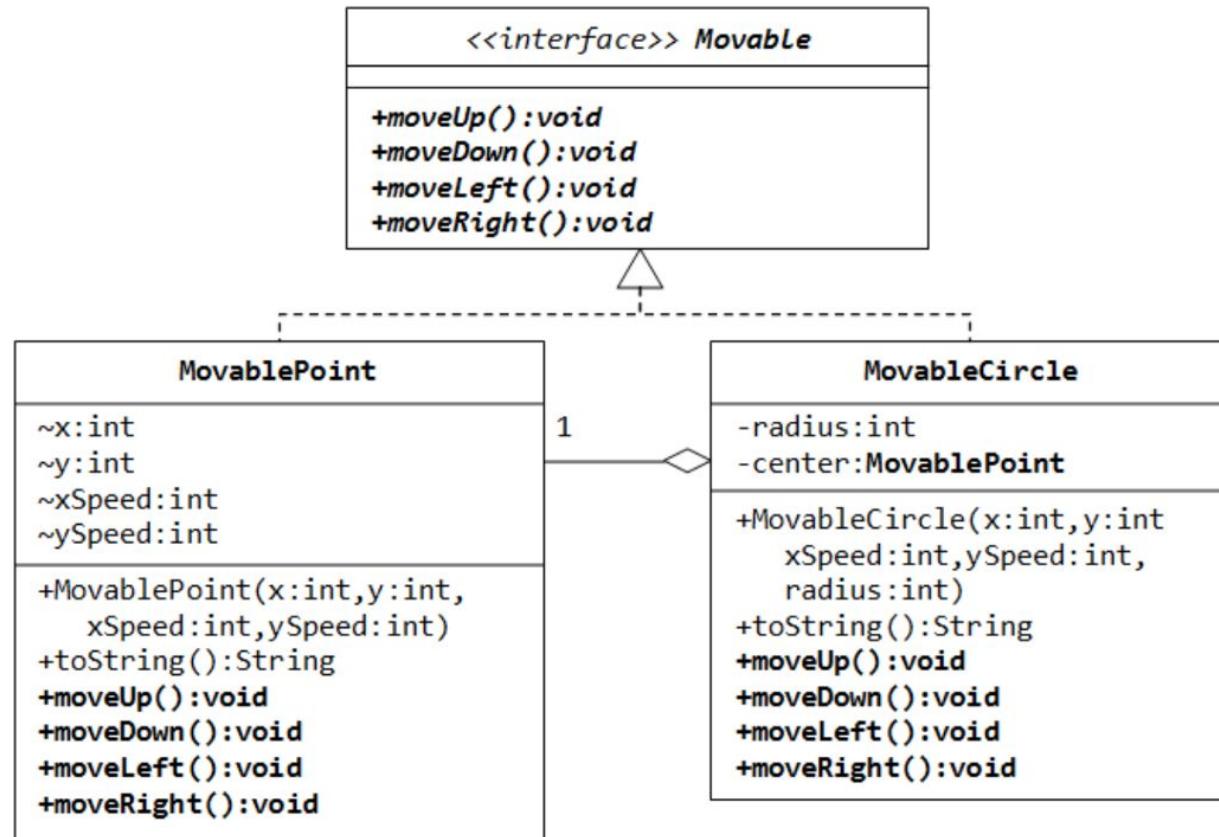


DESIGN

- Above we have an animal parent class with all public member fields.
- You can see the arrows originating from the duck, fish, and zebra child classes which indicate they inherit all the members from the animal class.
- Not only that, but they also implement their own unique member fields.
- You can see that the duck class has a swim() method as well as a quack() method.

Interface

- A relationship between two model elements, in which one model element implements/executes the behavior that the other model element specifies.

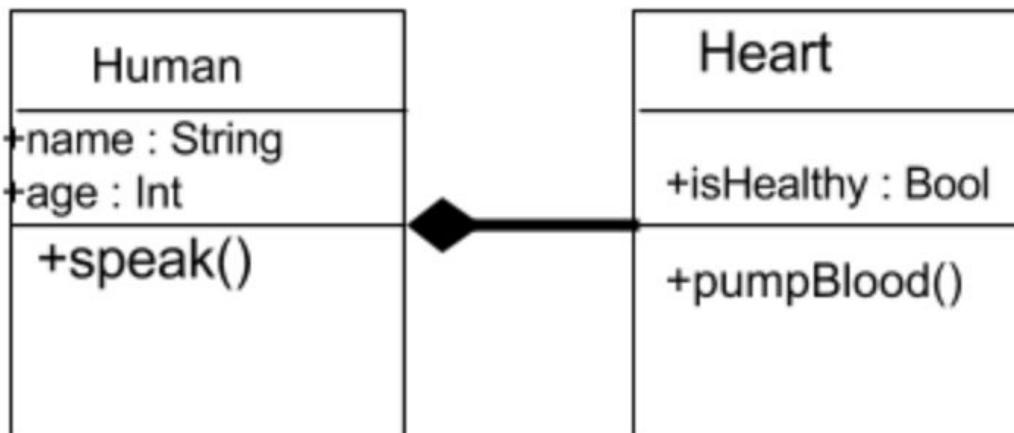


Dependency - Aggregation

- A special form of association which is a unidirectional (a.k.a one way) relationship between classes.
- The best way to understand this relationship is to call it a “has a” or “is part of” relationship.
- For example, consider the two classes: Wallet and Money.
- A wallet “has” money. But money doesn’t necessarily need to have a wallet so it’s a one directional relationship.

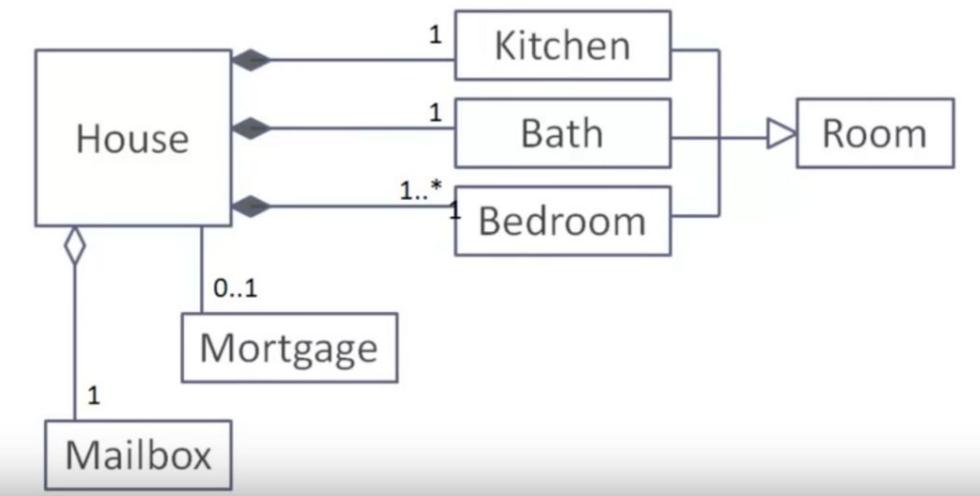
Composition

- A restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other.
- A human needs a heart to live and a heart needs a human body to function on. In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one does too) then it's a composition.



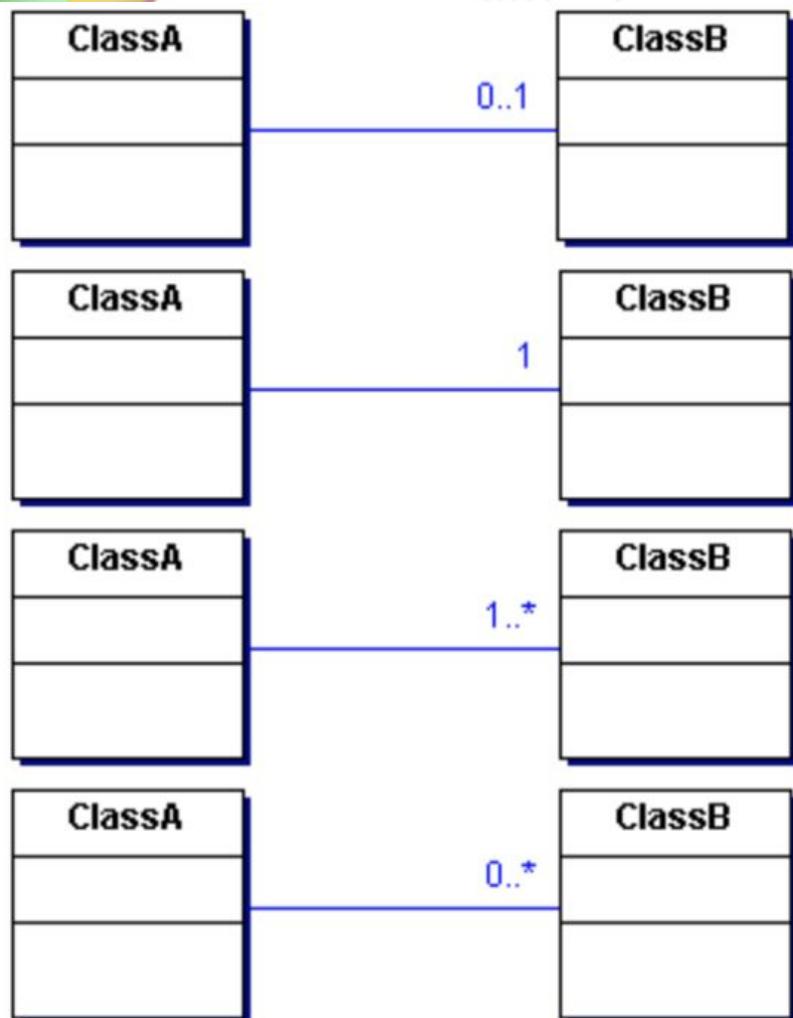
Multiplicity

- After specifying the type of association relationship by connecting the classes, you can also declare the cardinality between the associated entities. For example:



- The above UML diagram shows that a house has exactly one kitchen, exactly one bath, at least one bedroom (can have many), exactly one mailbox, and at most one mortgage (zero or one).

DESIGN



Objects of ClassA MAY
know about a single
object of ClassB

Objects of ClassA MUST
know about a single
object of ClassB

Objects of ClassA MUST
know at least one object
of ClassB

Objects of ClassA MAY
know about many objects
of ClassB

UML Class Diagram Example

- Now, let's take what we've learned in the previous tutorial and apply it.
- In this example we are asked to create a class diagram for a banking system. It must have the following classes:
 - Bank
 - ATM
 - Customer
 - Account
 - Transaction
 - Checking Account
 - Savings Account

Determine Possible Class Members Bank

- The bank class represents a physical bank.
- It has a location and a unique id.
- This bank also manages several accounts. **There's an association!**
 - What type of association is this?
 - Is a bank entirely composed of accounts (composition)?
 - Or are accounts 'part of' a bank (aggregation)?
 - It looks like aggregation.
 - It can't be composition because that would mean that both classes live and die together.
 - That's not quite right because you can have a bank without accounts and you can have accounts without a bank.
- We'll add a method called `getAccounts()`.

ATM

- The ATM class represents a physical ATM.
- Right off the bat, we can come up with three methods for the ATM:
 - withdraw()
 - deposit()
 - checkBalance()
- Each of these methods takes the card number as input.
- In terms of attributes, an ATM has a location and is managed by a specific bank.

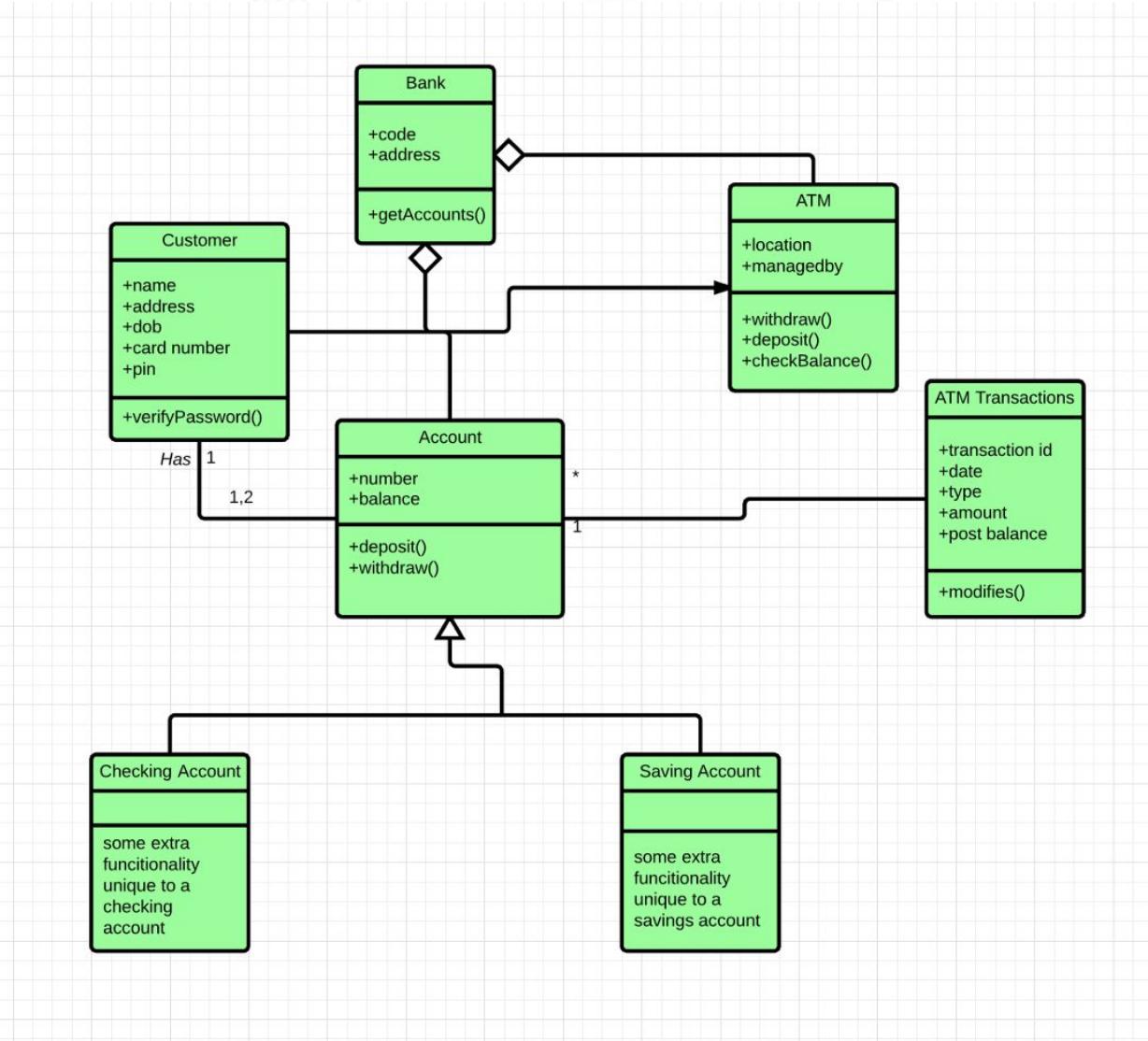
Customer

- The customer class represents a real customer.
- This customer has a
 - name
 - address
 - date of birth (dob)
 - card number
 - pin
- For this person to be considered a customer, they must have an account. **There's another association!
 - This isn't aggregation or composition, it's just a bi-directional association (drawn using a blank line no arrows).

Account

- The account class represents a bank account.
- Common attributes of bank accounts include
 - account number
 - balance
 - And more
- You can deposit() withdraw() money from the account.
- In addition, banks might offer two types of accounts:
 - A checking account
 - A savings account.
 - These two can thus be considered child classes of the account class and can inherit from it too.
 - We'll denote this by using a solid black line with an unfilled arrow going into the account class.

Completed Diagram



Credits

- **UML Class Diagrams Tutorial, Step by Step by Salma**
- https://medium.com/@smagid_allThings/uml-class-diagrams-tutorial-step-by-step-520fd83b300b

Object Oriented Programming with C++(203105337)

Prof. Sheetal Pandya, Assistant Professor
Computer science & Engineering



CHAPTER-7

I/O And File Management



C++ Basic I/O

C++ language provides a set of standard built-in functions which will do the work of reading and displaying data or information on the I/O devices during program execution. Such I/O functions establish an interactive communication between the program and user. C++ I/O occurs in streams, which are sequences of bytes.

If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation**.

If bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**.

Example

```
int main ()  
{  
int var1;  
char var2[10];  
cout << "Address of var1 variable: "; cout << &var1 << endl;  
cout << "Address of var2 variable: "; cout << &var2 << endl;  
return 0;  
}
```

Output : Address of var1 variable: 0x7ffd931243dc

Address of var2 variable: 0x7ffd931243d2



Header File

There are following header files important to C++ programs –

Sr.No	Header File & Function and Description
1	<iostream> This file defines the cin , cout , cerr and clog objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
2	<iomanip> This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as setw and setprecision .
3	<fstream> This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter.

The Standard Output Stream (cout)

The predefined object cout is an instance of ostream class.

The cout object is said to be "connected to" the standard output device, which usually is the display screen.

The cout is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example..

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

```
int main()
```

```
{
```

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

```
cout << "Value of str is : " << str << endl;
```

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

Value of str is : Hello C++



The Standard input Stream (cin)

The predefined object cin is an instance of istream class.

The cin object is said to be attached to the standard input device, which usually is the keyboard.

The cin is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example

```
#include <iostream> int main()
{
char name[50];
cout << "Please enter your name: "; cin >> name;
cout << "Your name is: " << name << endl;
}
```

When the above code is compiled and executed, it will prompt you to enter a name . You enter a value and then hit enter to see the following result – Please enter your name: cplusplus . Your name is: cplusplus



The Standard Error Stream (cerr)

The predefined object **cerr** is an instance of **ostream** class.

The **cerr** object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to **cerr** causes its output to appear immediately.

The **cerr** is also used in conjunction with the stream insertion operator as shown in the following example.

```
int main()
{
char str[] = "Unable to read....";
cerr << "Error message : " << str << endl;
}
```

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



Error message : Unable to read...

The Standard Error Stream (cerr)

The predefined object **clog** is an instance of **ostream** class.

The **clog** object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered.

This means that each insertion to **clog** could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

The **clog** is also used in conjunction with the stream insertion operator as shown in the following example.

```
int main() {  
    char str[] = "Unable to read....";  
    clog << "Error message : " << str << endl;  
}
```

When the above code is compiled and executed, it produces the following result – Error message : Unable to read.



C++ file and streams

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

1. ofstream-This data type represents the output file stream and is used to create files and to write information to files.

2. ifstream-This data type represents the input file stream and is used to read information from files.

3. fstream-This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.



Opening File

A file must be opened before you can read from it or write to it.

Either **ofstream** or **fstream** object maybe used to open a file for writing.

And ifstream object is used to open a file for reading purpose only.

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

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

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



Closing File

- When program terminates it automatically a c++ streams, release all the allocated flushes memory and close all the opened files.
- it is always a good practice that a programmer should close all the opened files before program termination.
- Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.
- void close();



Writing & Reading File

Writing File: While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen.

The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading File: You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard.

The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.



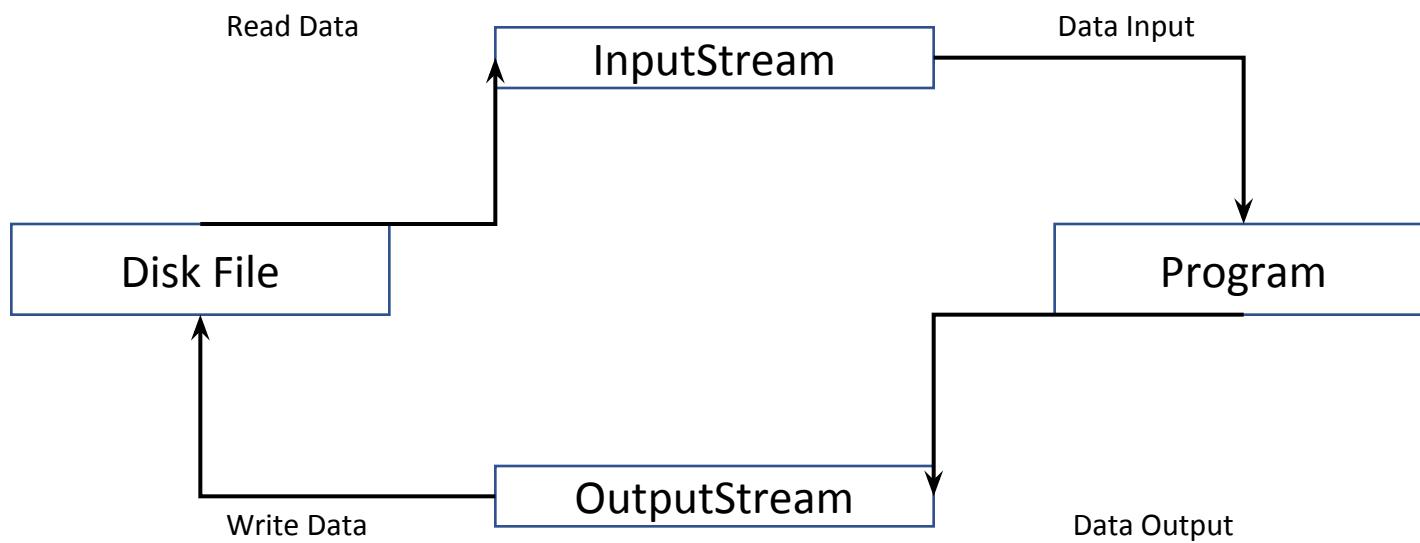
Manipulator are show in below table

Manipulators are used to manipulate the output in specific format.

Manipulators	Use
<code>setw()</code>	To specify the required field size for displaying an output value.
<code>setprecision()</code>	To specify the number of digits to be displayed after the decimal point of a float value.
<code>setfill()</code>	To specify a character that is used to fill the unused portion of a field.
<code>setiosflags()</code>	To specify format flags that can control the form of output display such as left or right justification.
<code>resetiosflags()</code>	To clear the flags specified.

File Stream

- A File Stream act as an interface between program and the files.
- The Stream that supplies data to the program is know as input Stream.
- The Stream that receives data from the program is know as output Stream.



File Stream Classes

C++ provides the following classes to perform output and input of characters to files:

- ifstream class represents the input disk(read) file
- ofstream class represents the output disk(write) file
- fstream class represents both disk(read and write) file



File Management Function

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



File Mode

- File modes allow us to *create, read, write, append or modify* a file. The file modes are defined in the class "ios"
- ios::in :- (**input**) Searches for the file and opens it in the read mode only. If the file is found.(for input operations)
- ios::out :- (**output**) Searches for the file and opens it in the write mode. If the file is found, its content is overwritten. If the file is not found, a new file is created. (for output operations)
- ios::app :- (**append**) Searches for the file and opens it in the append mode. All output operations are performed at the end of the file, appending the content to the current content of the file.
- ios::binary :- (**binary**) Searches for the file and opens the file, if the file is found in a binary mode to perform binary input or output file operations.



File Mode

- ios::ate :- (**at end**) Searches for the file, and opens it for output and moves the read or write control to the end at the file.
- ios::trunc :- (**truncate**) Searches for the file and opens it to truncate or deletes all of its content, If that file exists.



Thank you



state diagram

A **state diagram** is used to represent the condition of the system or part of the system at finite instances of time. It's a **behavioral** diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams**. These terms are often used interchangeably. So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli. We can say that each and every class has a state but we don't model every class using State diagrams. We prefer to model the states with three or more states.

Uses of statechart diagram –

- We use it to state the events responsible for change in state (we do not show what processes cause those events).
- We use it to model the dynamic behavior of the system .
- To understand the reaction of objects/classes to internal or external stimuli.

Firstly let us understand what are **Behavior diagrams**? There are two types of diagrams in UML :

1. **Structure Diagrams** – Used to model the static structure of a system, for example- class diagram, package diagram, object diagram, deployment diagram etc.
2. **Behavior diagram** – Used to model the dynamic change in the system over time. They are used to model and construct the functionality of a system. So, a behavior diagram simply guides us through the functionality of the system using Use case diagrams, Interaction diagrams, Activity diagrams and State diagrams.

Difference between state diagram and flowchart –

The basic purpose of a **state diagram** is to portray various changes in state of the class and not the processes or commands causing the changes. However, a **flowchart** on the other hand portrays the processes or commands that on execution change the state of class or an object of the class.

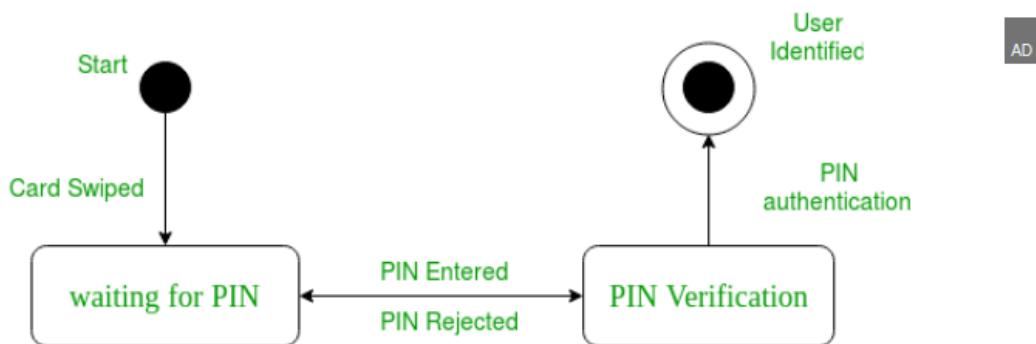


Figure – a state diagram for user verification

Figure – a state diagram for user verification

The state diagram above shows the different states in which the verification sub-system or class exist for a particular system.

Basic components of a statechart diagram –

1. **Initial state** – We use a black filled circle represent the initial state of a System or a class.

Initial state



- 2.

Figure – initial state notation

3. **Transition** – We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labelled with the event which causes the change in state.

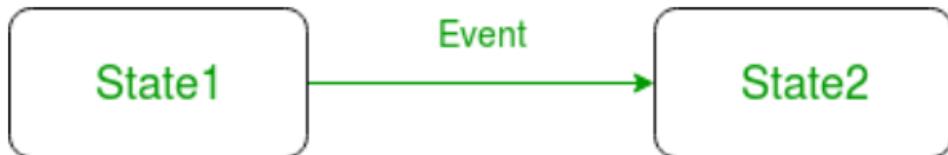


Figure – transition

4. **State** – We use a rounded rectangle to represent a state. A state represents the conditions or circumstances of an object of a class at an instant of time.



Figure – state notation

5. **Fork** – We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states.

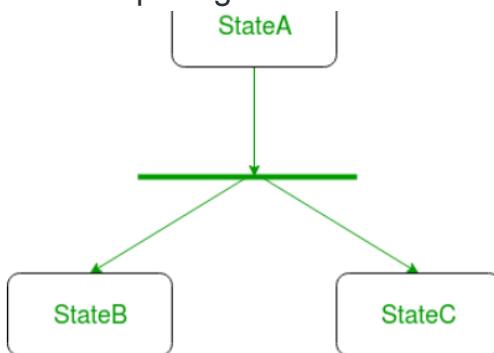


Figure – a diagram using the fork notation

6. **Join** – We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.

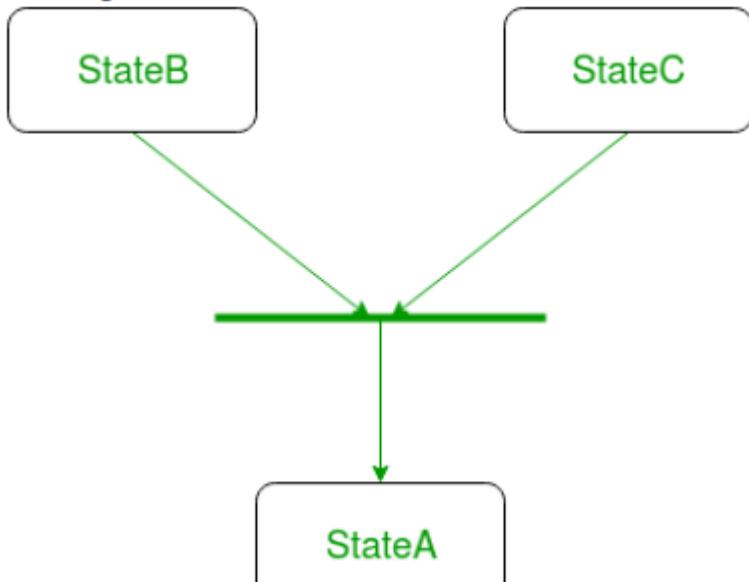


Figure – a diagram using join notation

7. **Self transition** – We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases.

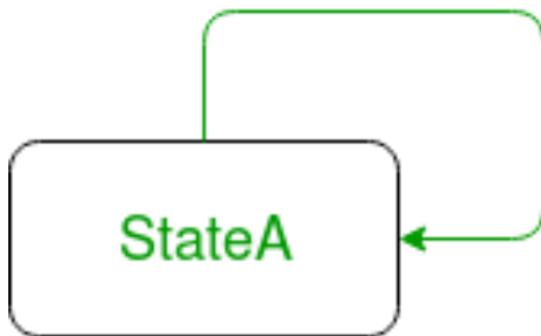


Figure – self transition notation

8. **Composite state** – We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state.

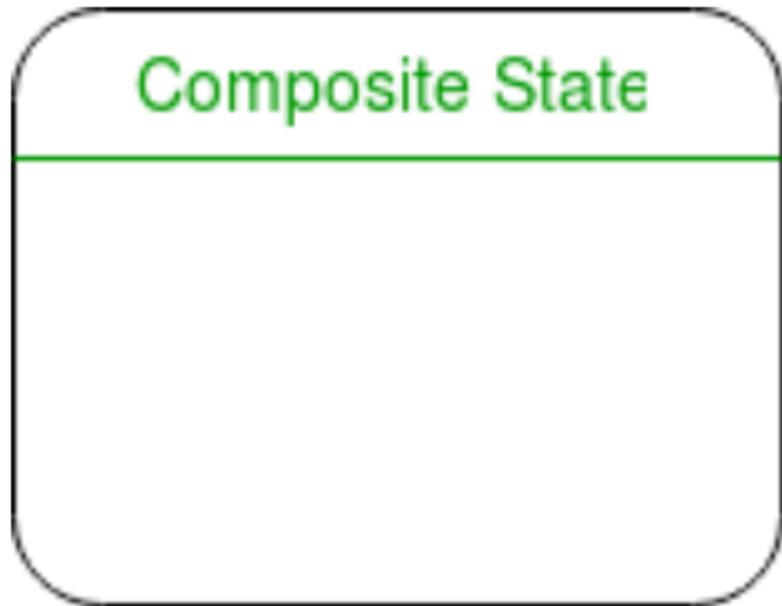


Figure – a state with internal activities

9. **Final state** – We use a filled circle within a circle notation to represent the final state in a state machine diagram.



Figure – final state notation

10.

Steps to draw a state diagram –

1. Identify the initial state and the final terminating states.

2. Identify the possible states in which the object can exist (boundary values corresponding to different attributes guide us in identifying different states).
3. Label the events which trigger these transitions.

Example – state diagram for an online order –

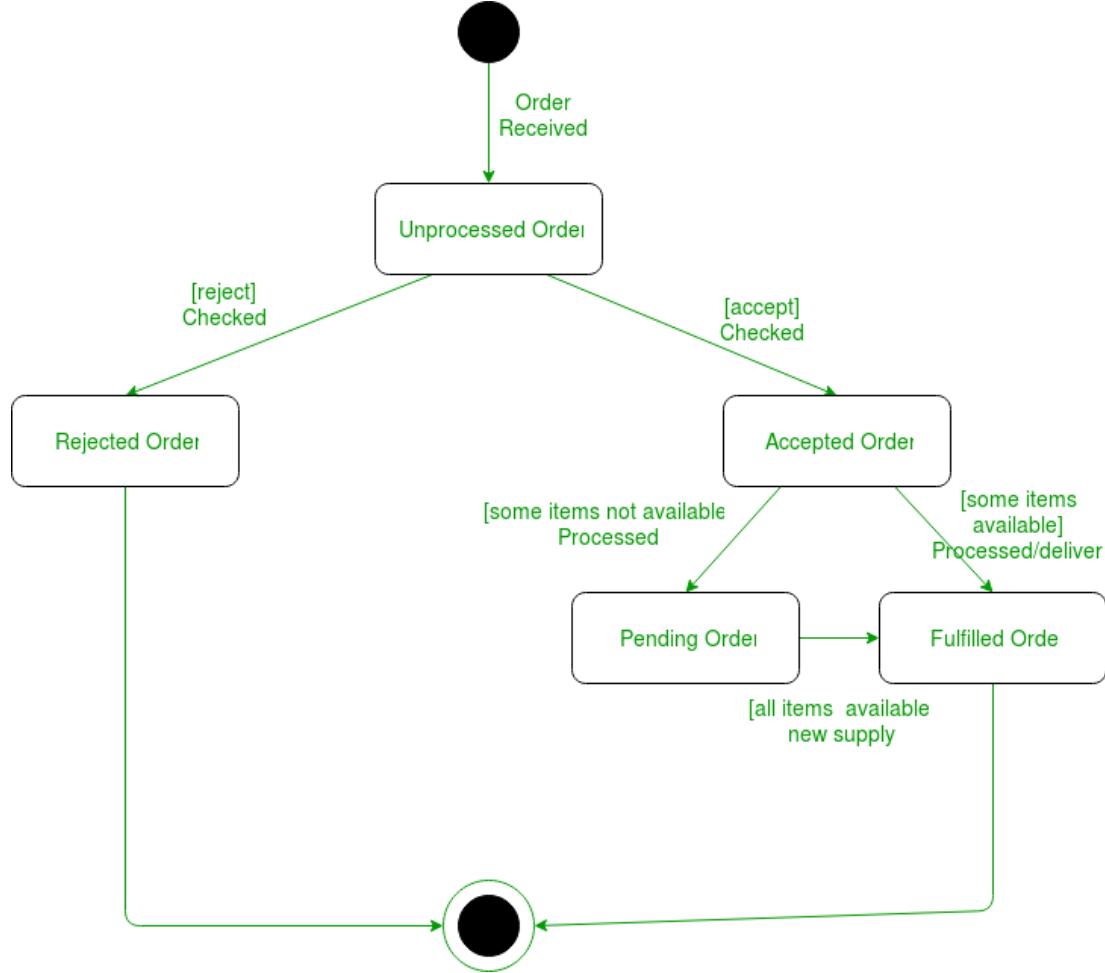


Figure – state diagram for an online order

The UML diagrams we draw depend on the system we aim to represent. Here is just an example of how an online ordering system might look like :

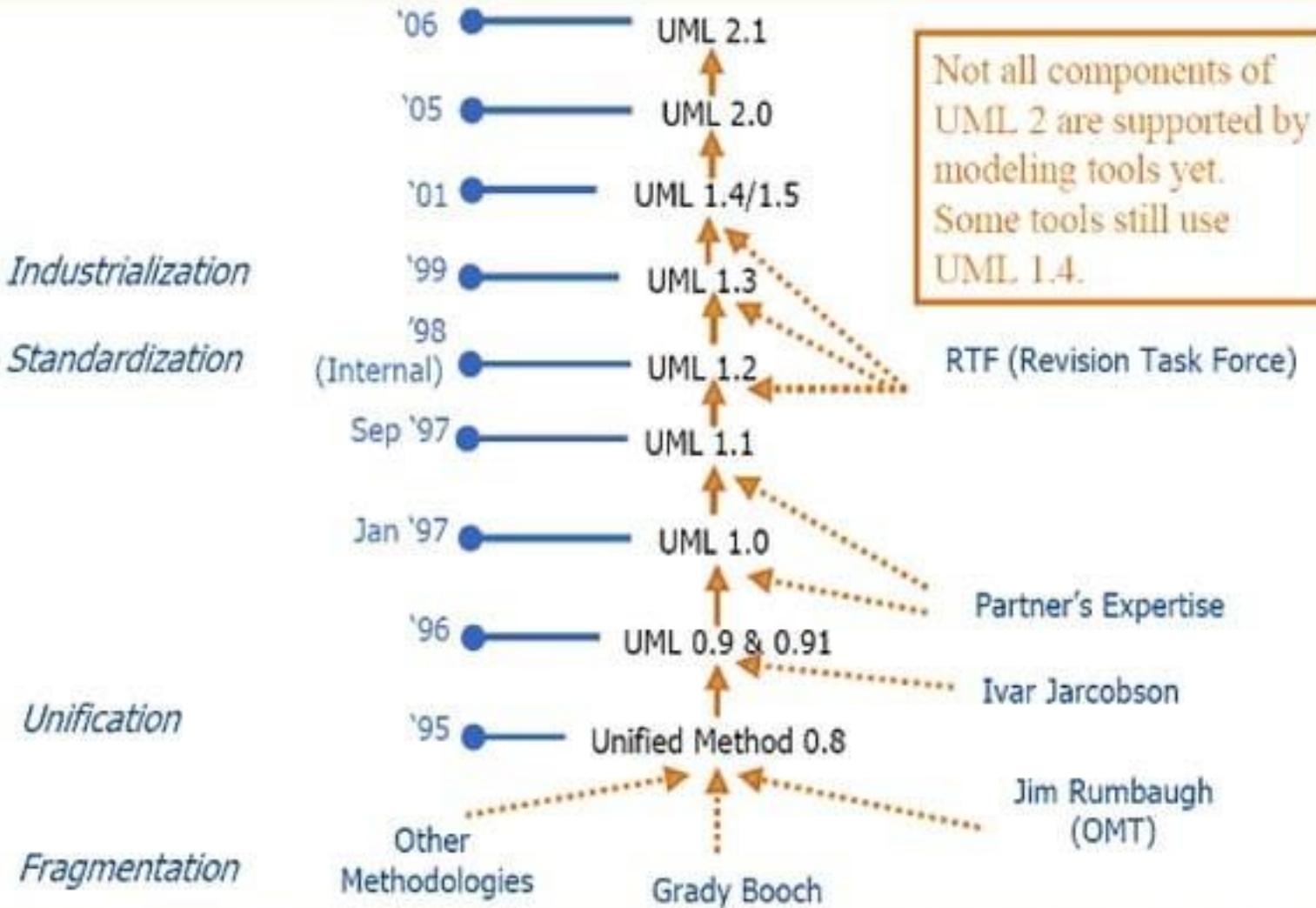
1. On the event of an order being received, we transit from our initial state to Unprocessed order state.
2. The unprocessed order is then checked.
3. If the order is rejected, we transit to the Rejected Order state.
4. If the order is accepted and we have the items available we transit to the fulfilled order state.
5. However if the items are not available we transit to the Pending Order state.

6. After the order is fulfilled, we transit to the final state. In this example, we merge the two states i.e. Fulfilled order and Rejected order into one final state.

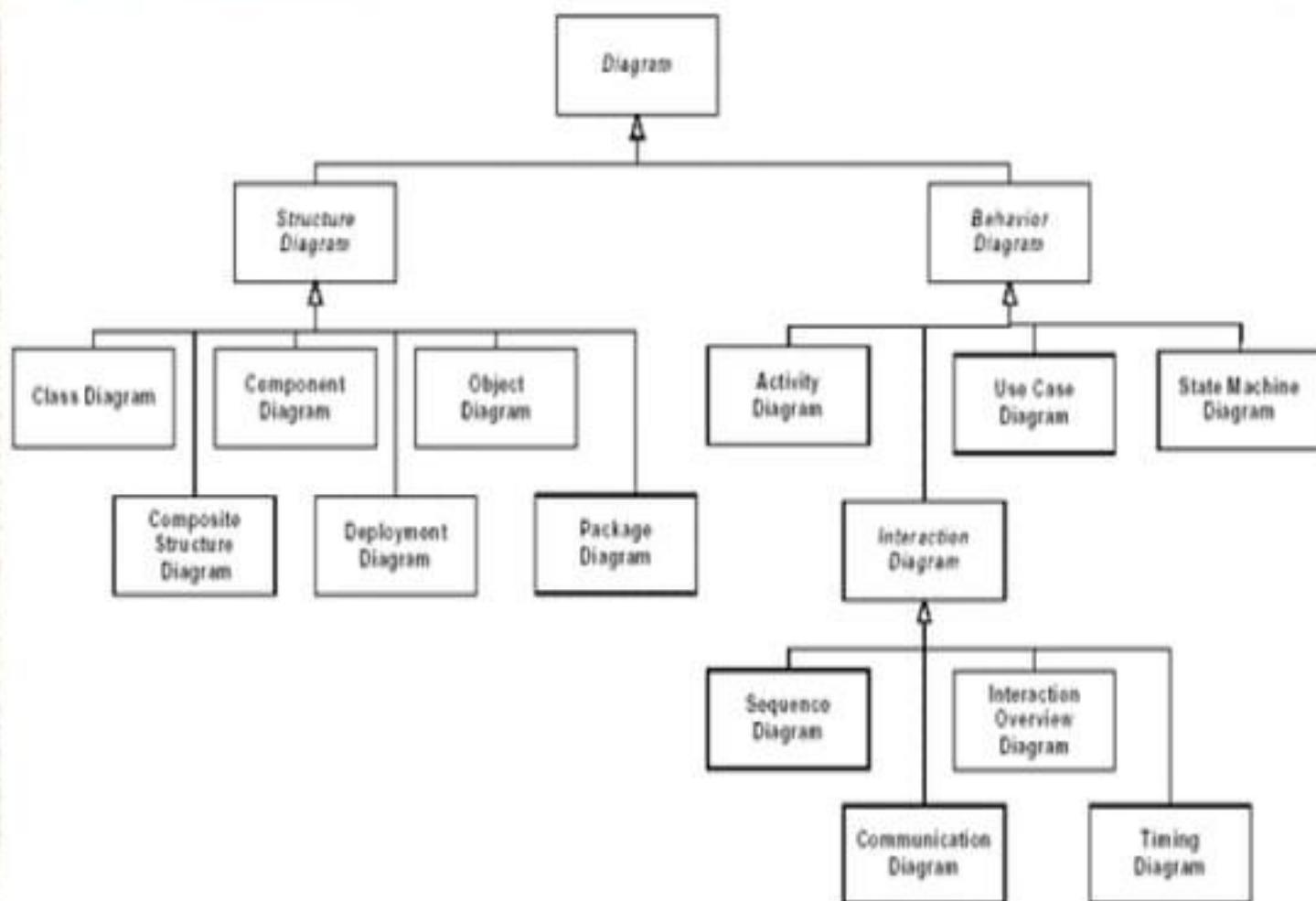
Introduction

- UML - language and notation for specification, construction, visualisation and documentation of models of software systems.
- UML is not:
 - programming language
 - a CASE tool
 - method
- UML modelling language.
 - Methods: modelling language + process
 - Modelling language : Notation that methods use to express design
 - Process: Steps in doing a design
- Associate with UML is Unified Software Development Process.
- Object Management Group an industry standards body requested
- standard object modelling language (1996).
- UML: Developed by Grady Booch, Ivar Jacobson, James Rumbaugh

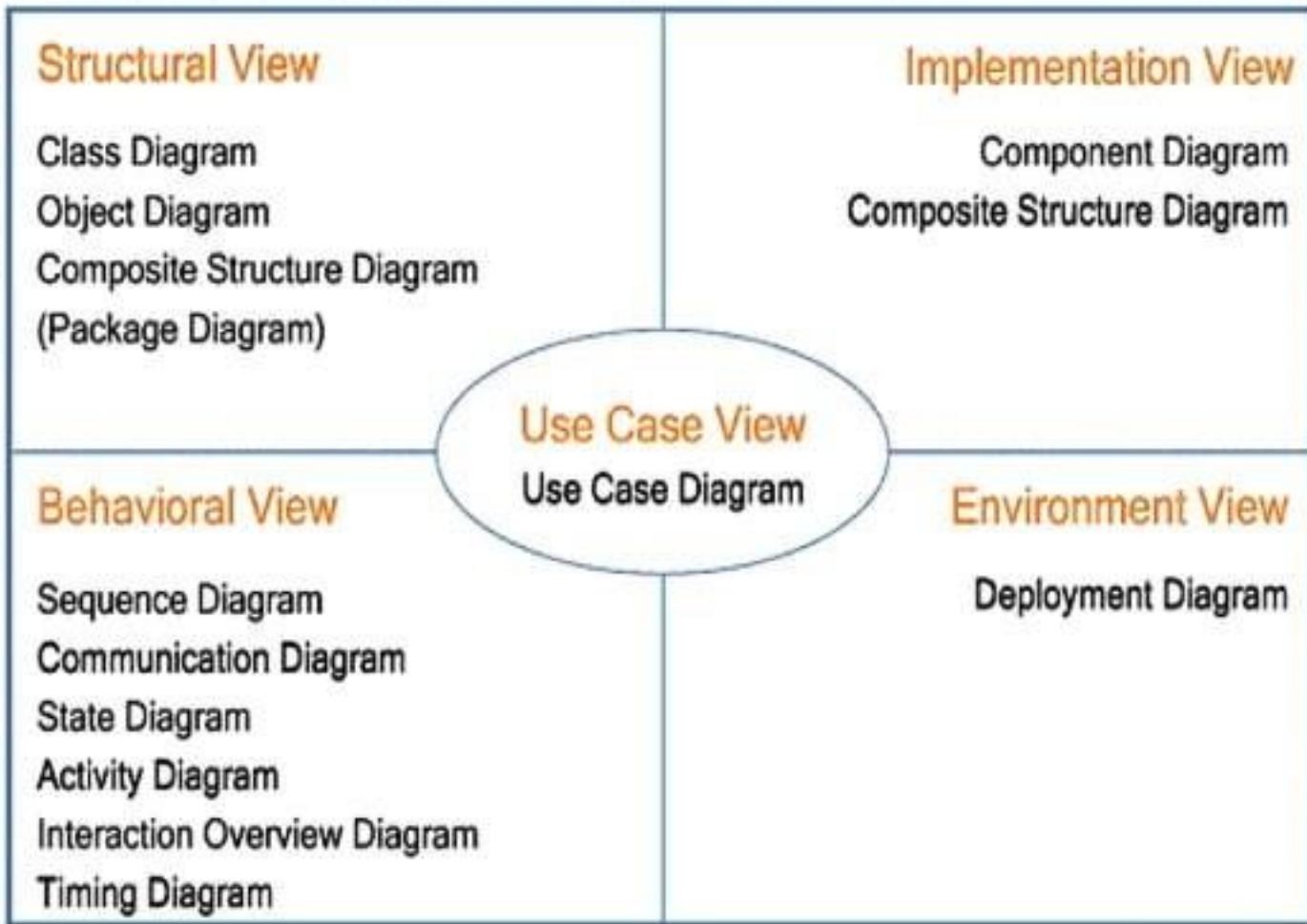
History



Diagrams



Diagrams



Use Case Diagrams

Use case:

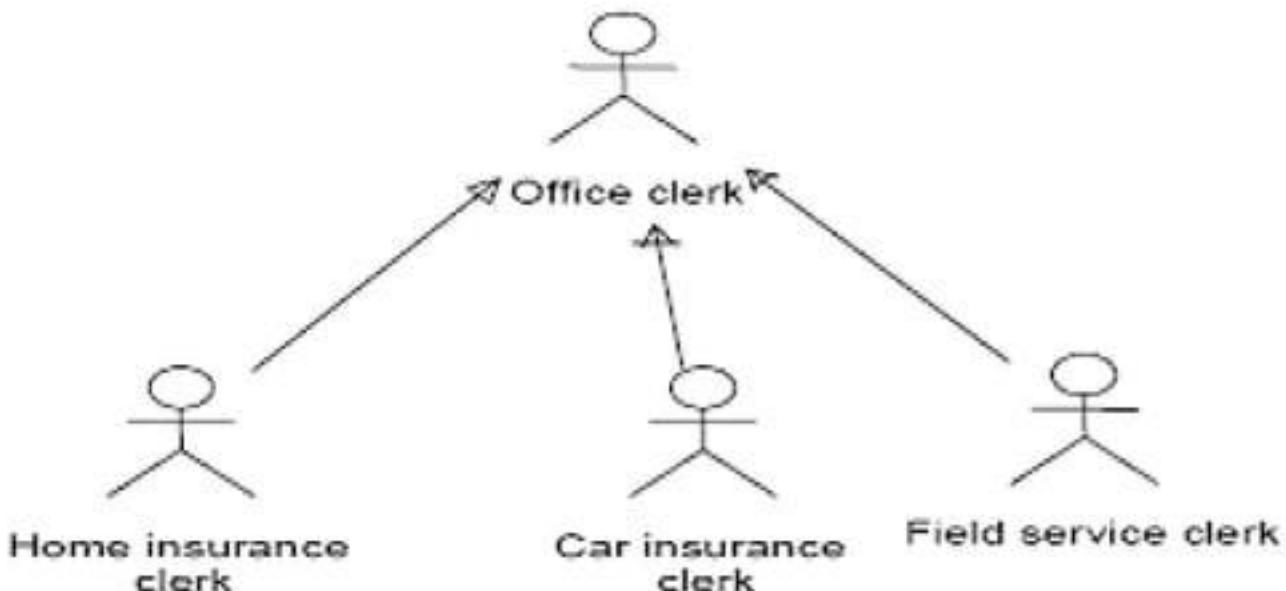
- Describes a task that a user can perform using the system.
- Describes requirements for the system
- Task described by a use case is composed of *activities*
- Use case can have different variations called *scenarios*
- Should not be used for functional decomposition !

Actors:

- Actor is an external entity which is involved in the interaction with the system described in a use case.
- Actors = roles
- Actors can be also dialogs, and external systems

Use Case Diagrams

- Generalisation and specialisation of actors

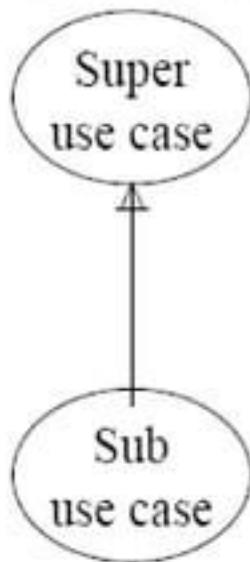
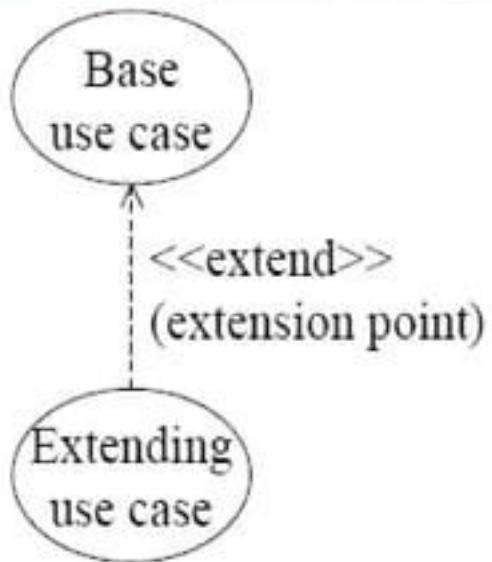
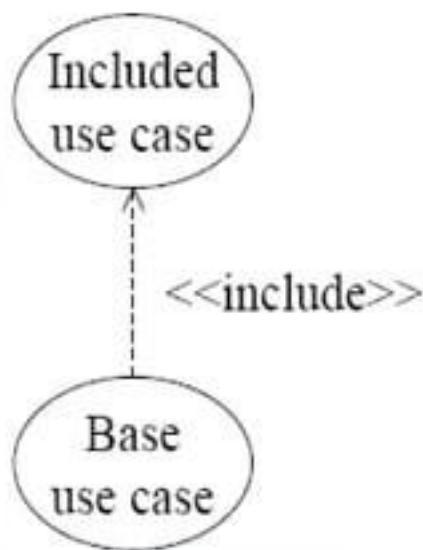


Use Case Diagrams

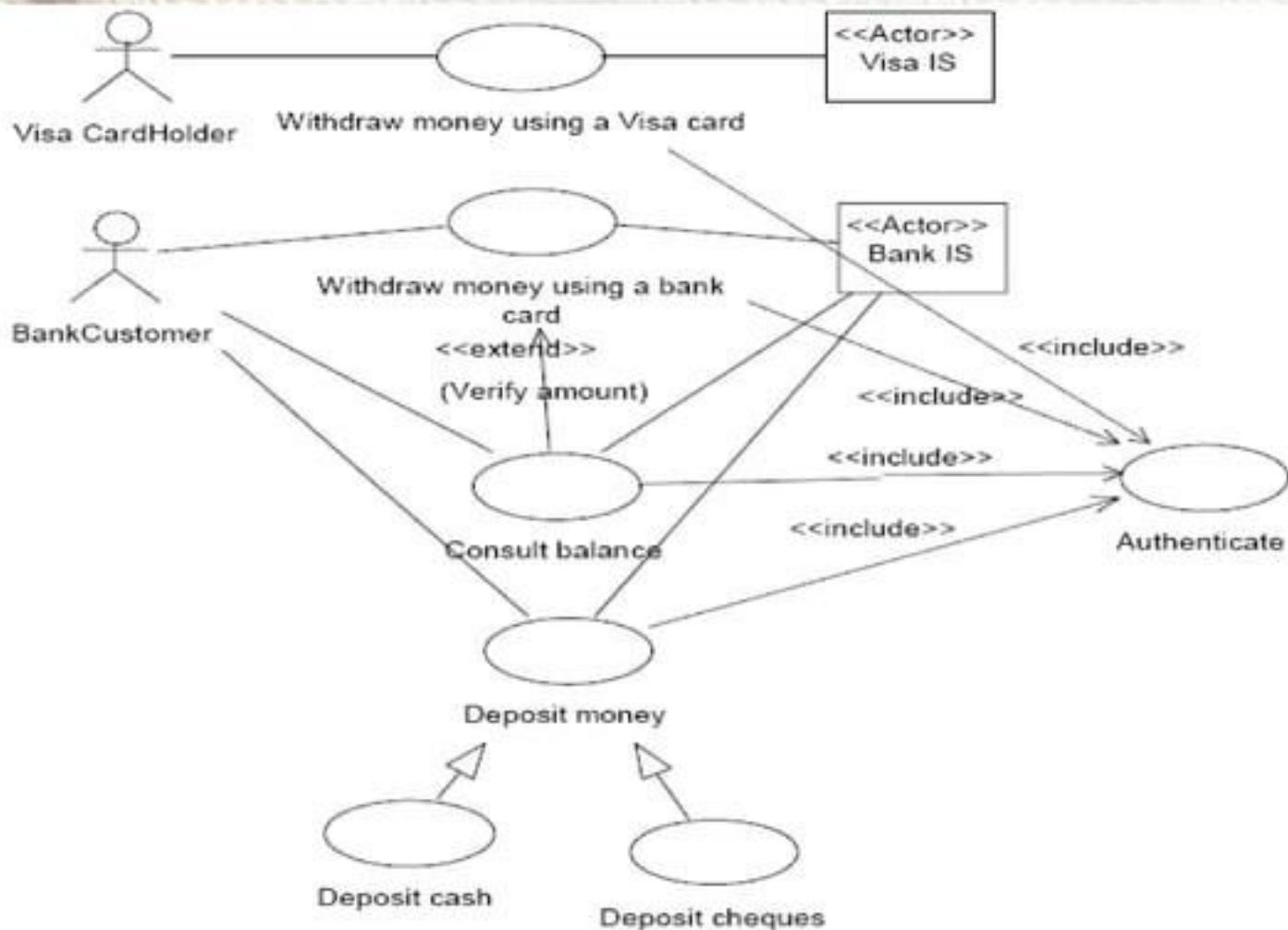
Use Case Diagrams

- Shows the relationships between a set of use cases and the actors involved in these use cases.
 - Tool for requirement determination
 - Use case describes those activities which are to be supported by the software under development
- Relationships between use cases
 - Include: base use case includes the functionality of included use case
 - Extend: a use case is optionally extended by functionality of another use case
 - Generalisation: sub use case inherits behaviour and semantics from super use cases

Use Case Diagrams



Use Case Diagrams



Class Diagram

Class Attributes:

- Represent named properties of a UML class
- Attribute declaration may include visibility, type and initial value: `+attributeName : type = initial-value`

Class Operations:

- Represent named services provided by a UML class
- Operation may include visibility, parameters, and return type: `+opName(param1 : type = initial_value) : return-type`

Class Visibility:

Three levels of class, attribute and operation visibility:

- Private (-), available only to the current class
- Protected (#), available to the current and inherited classes
- Public (+), available to the current and other classes

Class Diagram

Shape
+origin #width : int -height : int = 0
+move() #resize() : boolean -display(always : boolean = true) : boolean

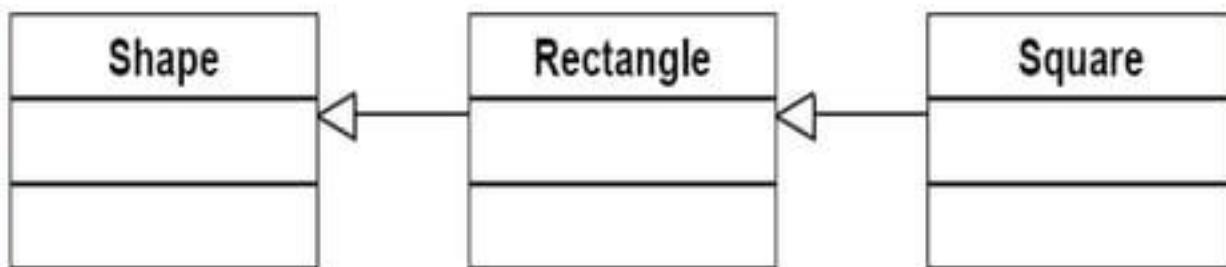
Class Objects:

Each class represents a set of objects that share the same attributes, operations, relationships, and semantics

s1 : Shape
origin = (10, 10)
width = 15
height = 30
...

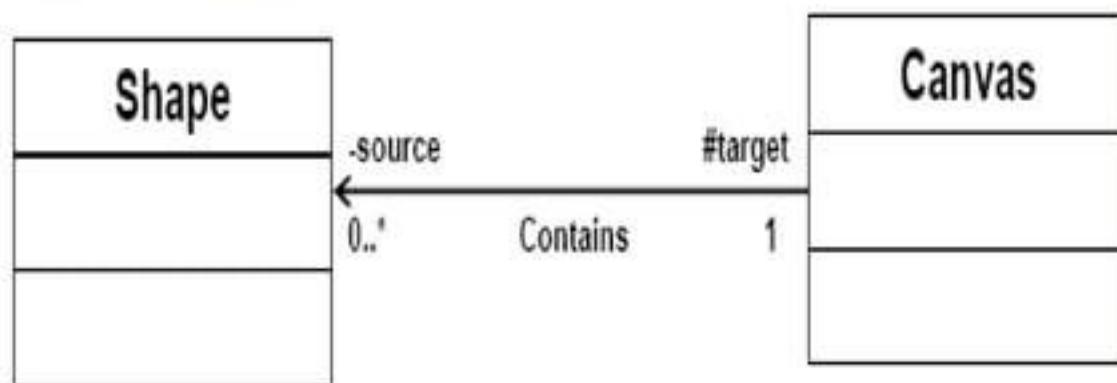
Class Generalization:

- Represent a relation between a parent (a more abstract class) and a child (a more specific class)
- Generally referred to as a “is-a-kind-of” relationship
- Child objects may be used instead of parent objects since they share attributes and operations; the opposite is not true



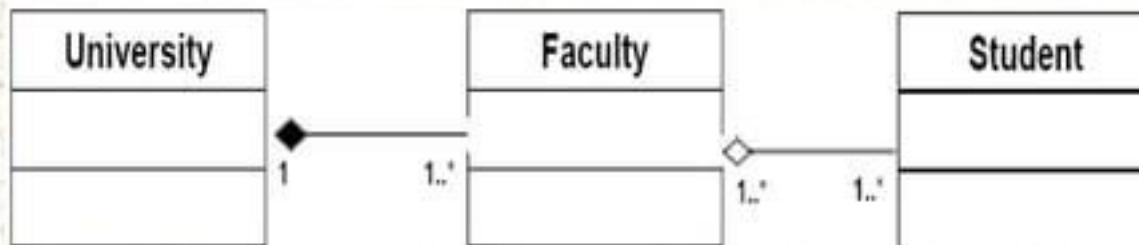
Class Association:

- Represent a structural relationship between class objects and may be used to navigate between connected objects
- Association can be binary, between two classes, or n-ary, among more than two classes
- Can include association name, direction, role names, multiplicity, and aggregation type



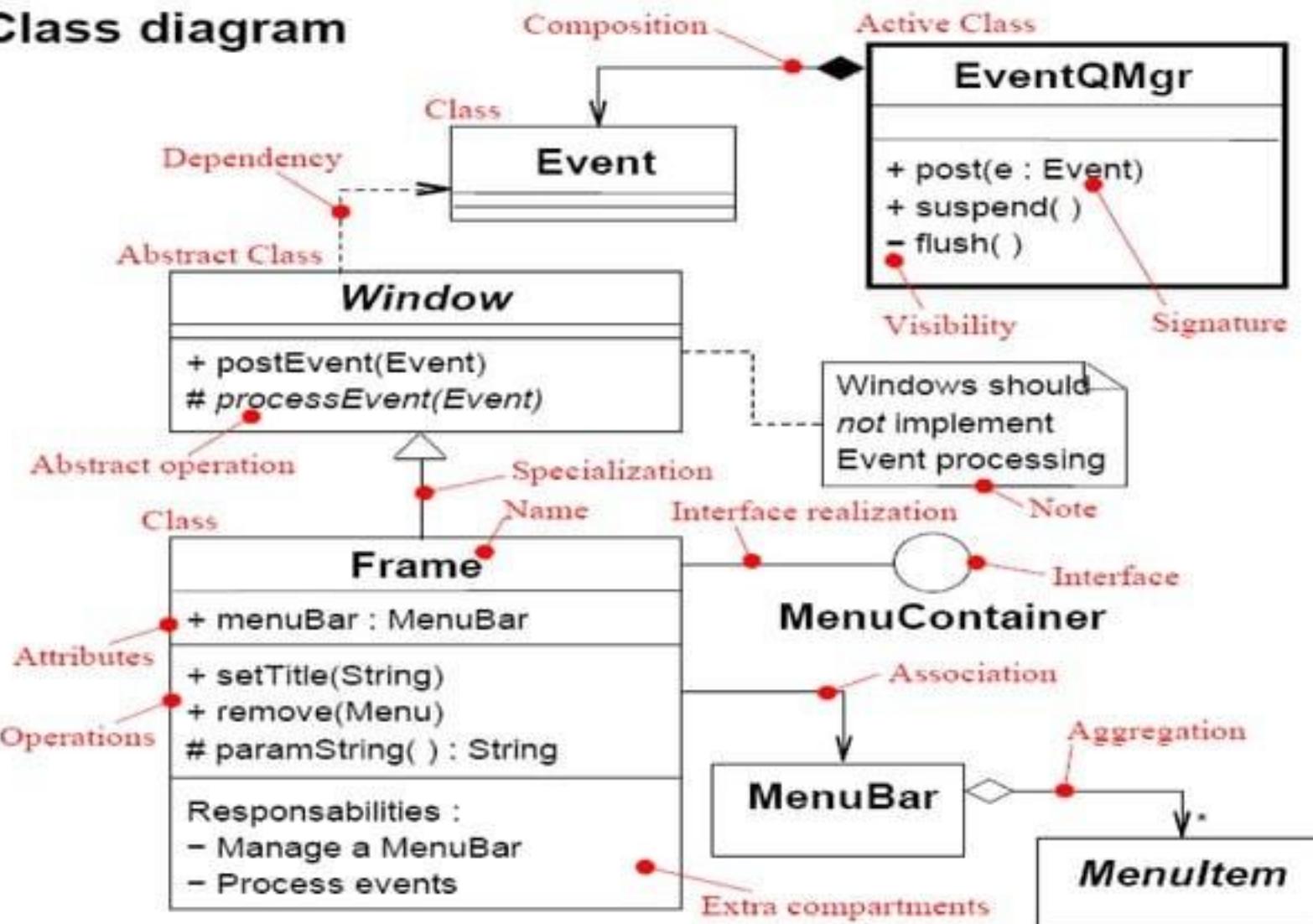
Class Aggregation

- Represent a specific, whole/part structural relationship between class objects
- Composition (closed diamond) represents exclusive relationship between two class objects (e.g., a faculty cannot exist without nor be a part of more than one university)
- Aggregation (open diamond) represents nonexclusive relationship between two class objects (e.g., a student is a part of one or more faculties)



Example

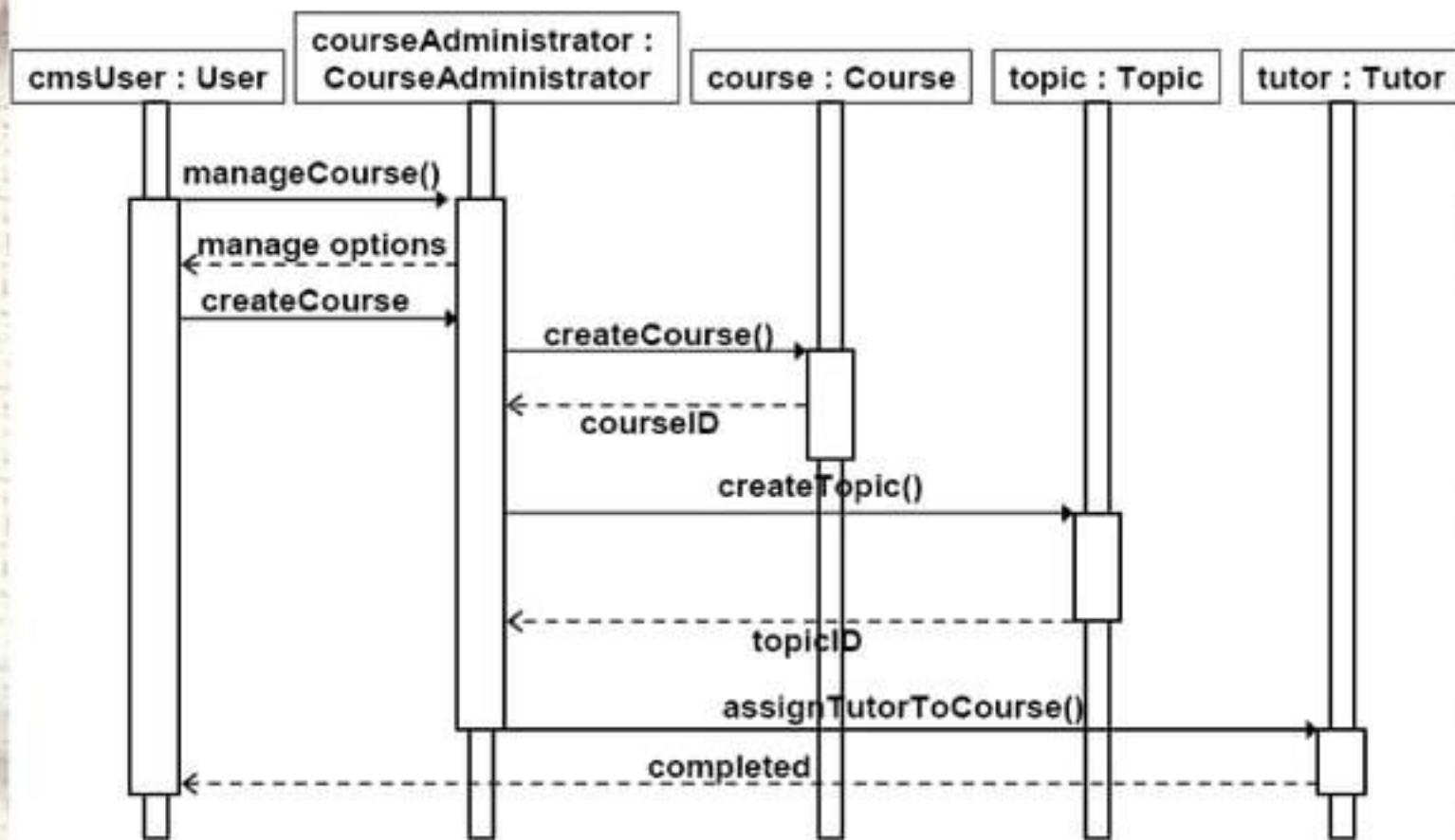
Class diagram



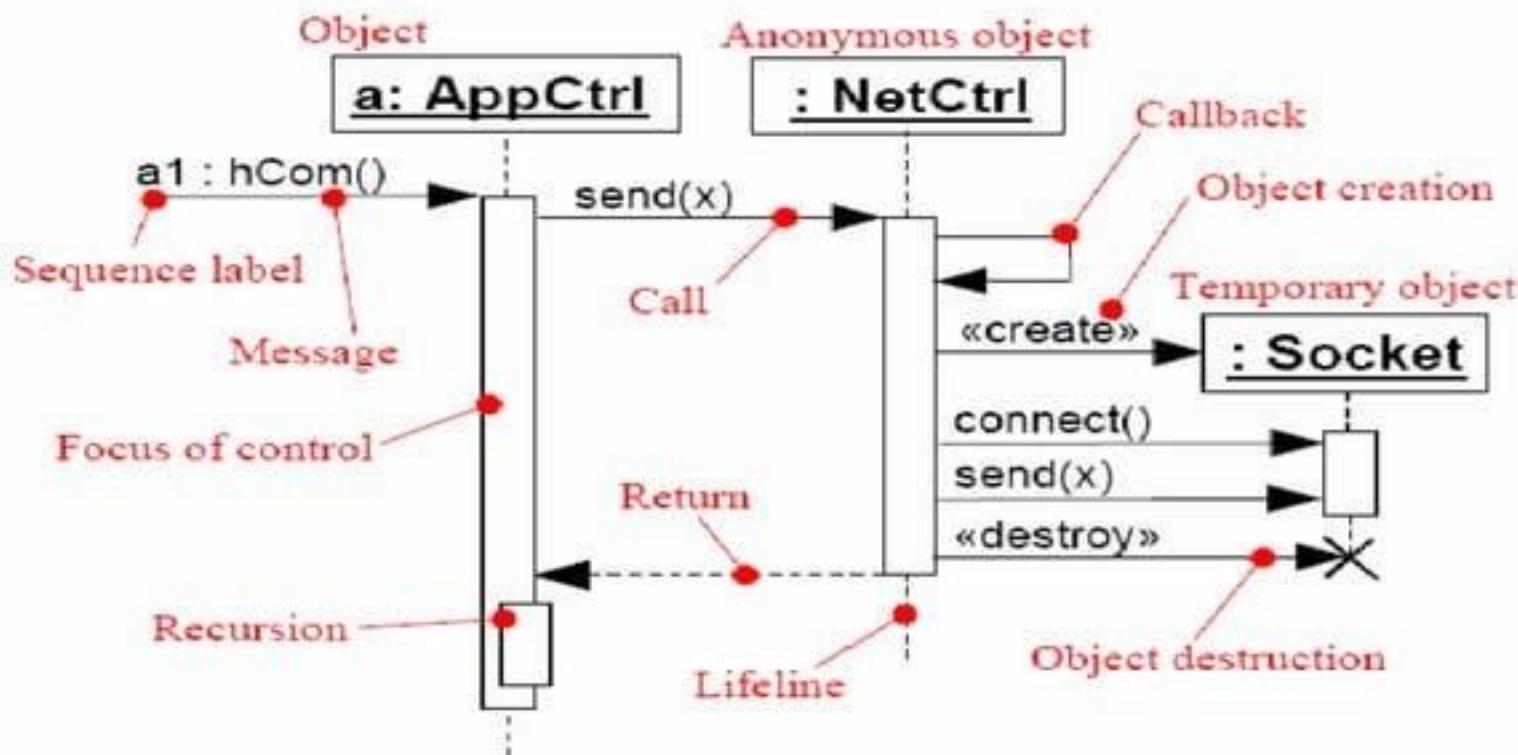
Interaction Diagrams

- Represent interaction between class objects based on conditions and operations
- Can also represent a use case scenario of interaction between actors and the system
- Two main subtypes: sequence and collaboration diagrams
- Sequence diagrams emphasize the temporal order of interaction and show lifetime of each object

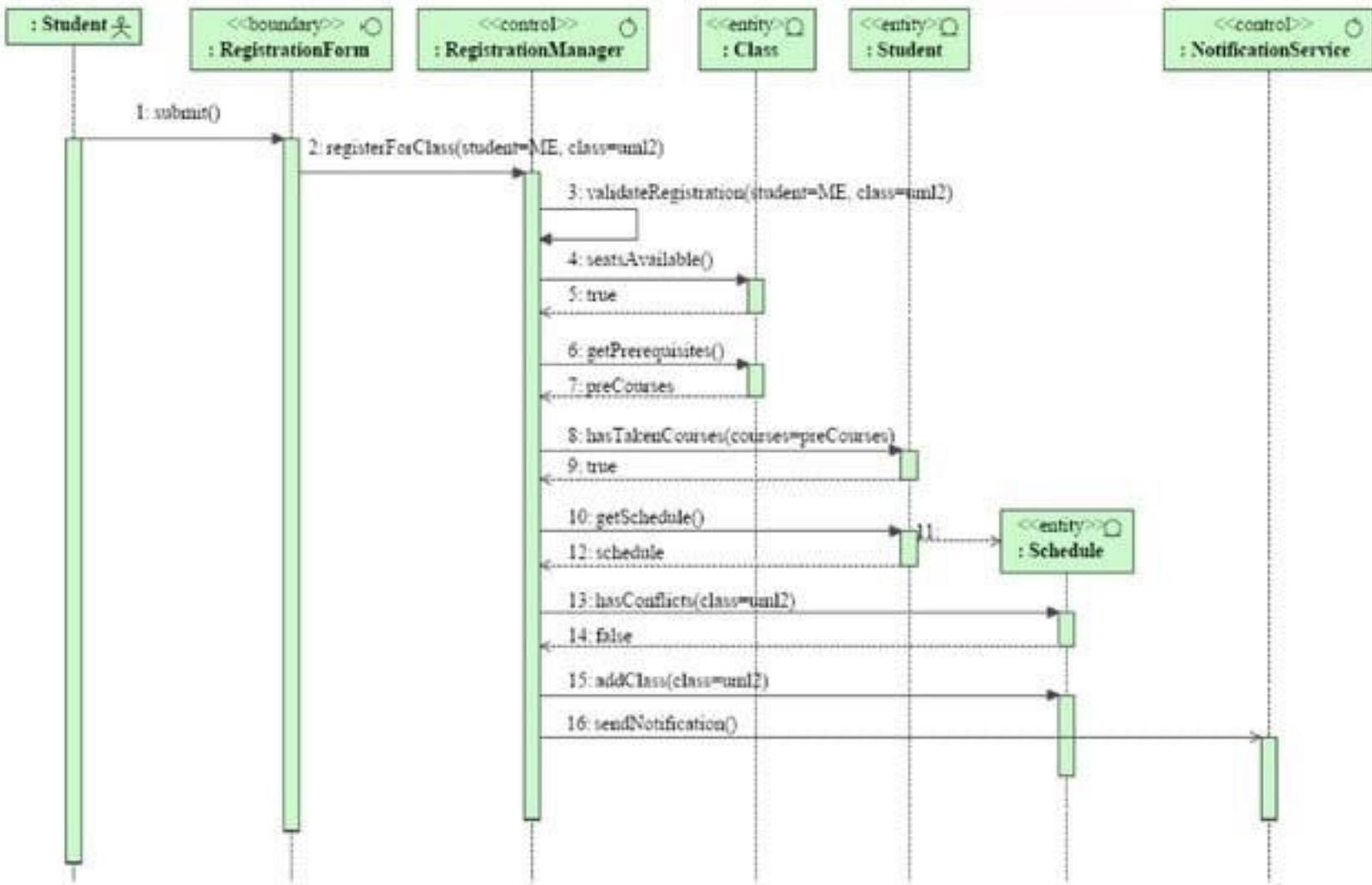
Sequence diagrams: example



Sequence diagrams: example

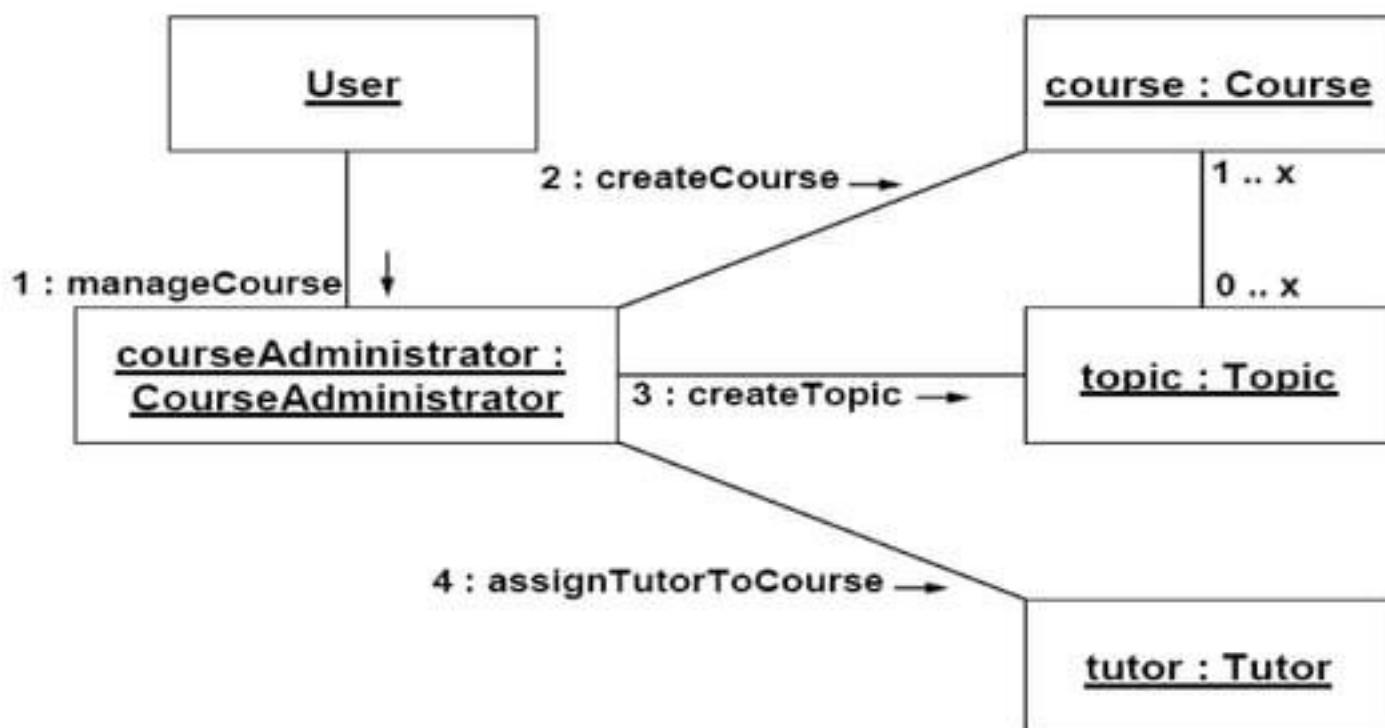


Sequence diagrams: example

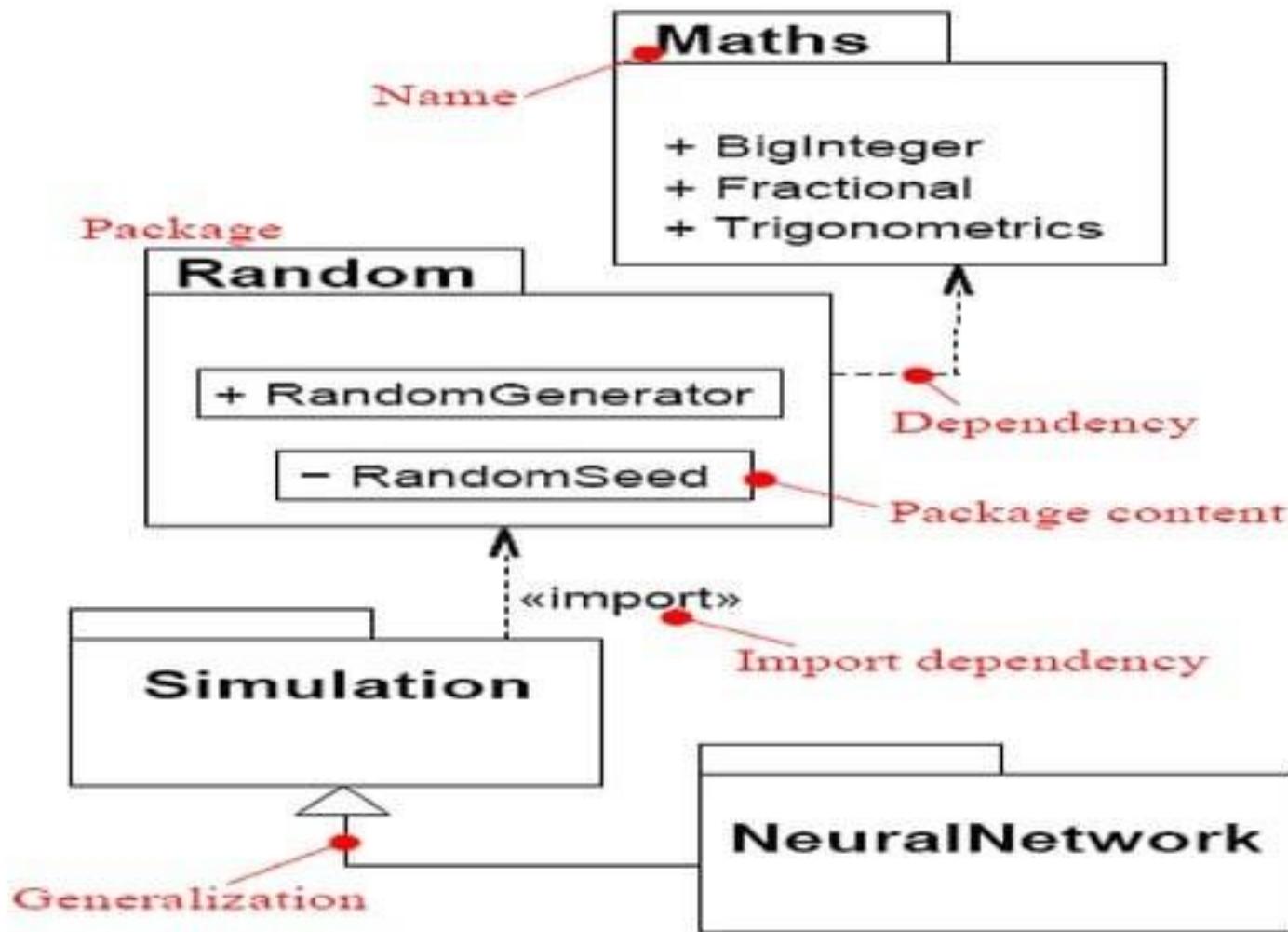


Collaboration diagrams

- Emphasize layout and show interaction as numbering of steps in a scenario

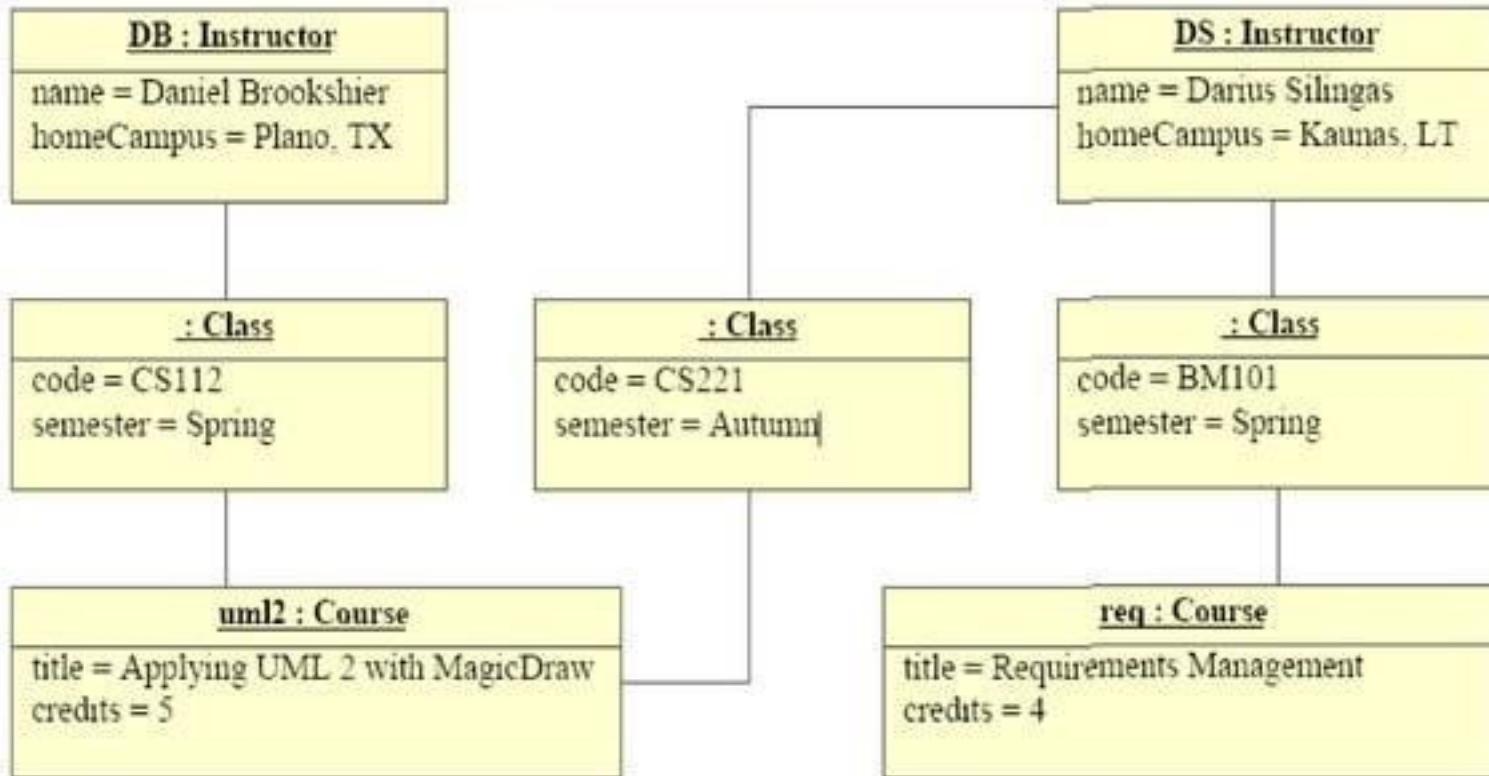


Collaboration diagrams: example



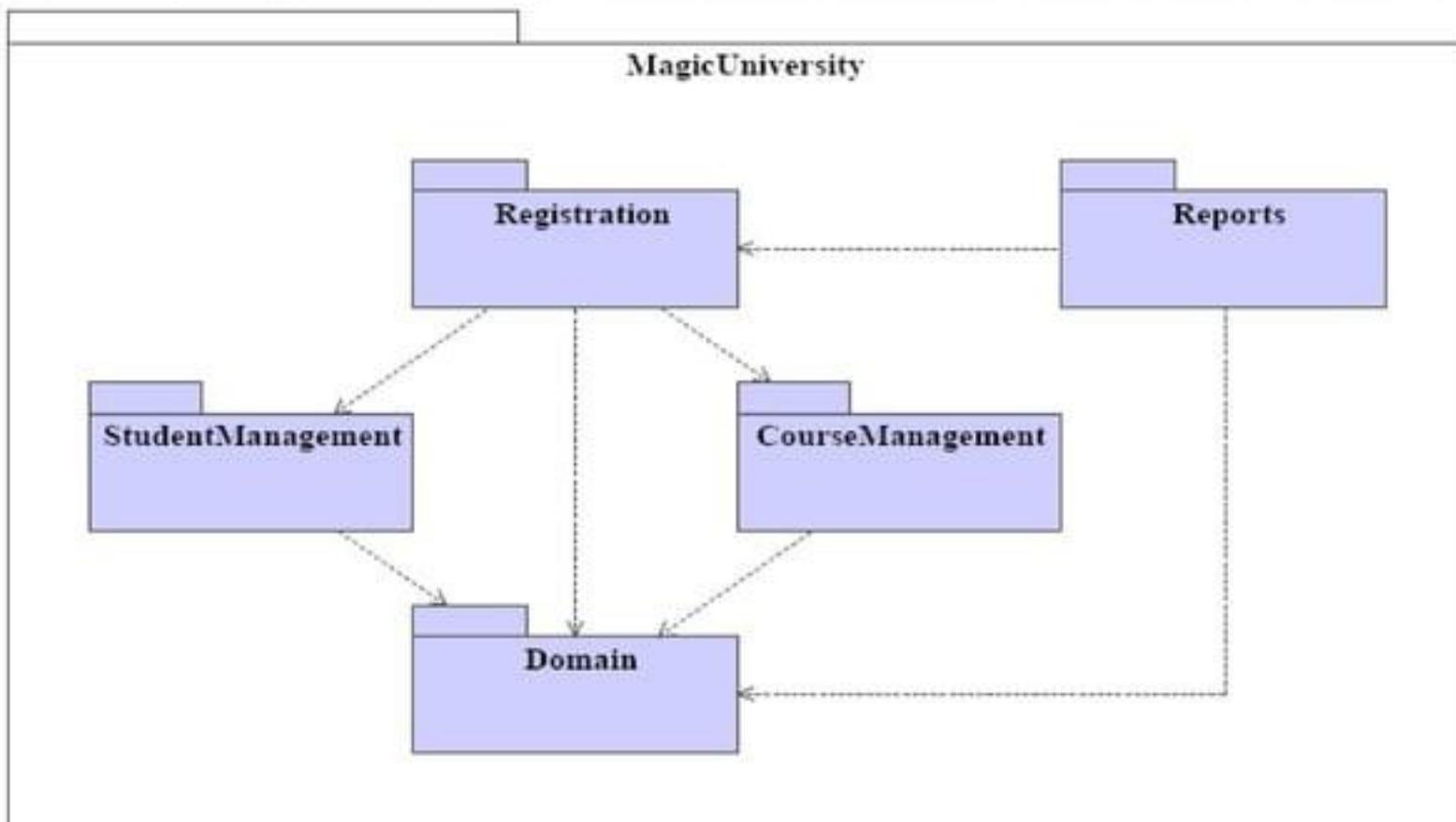
Object Diagram

- Shows an example of objects with slots and links that could be instantiated from defined classes and relationships
- Validates class diagrams



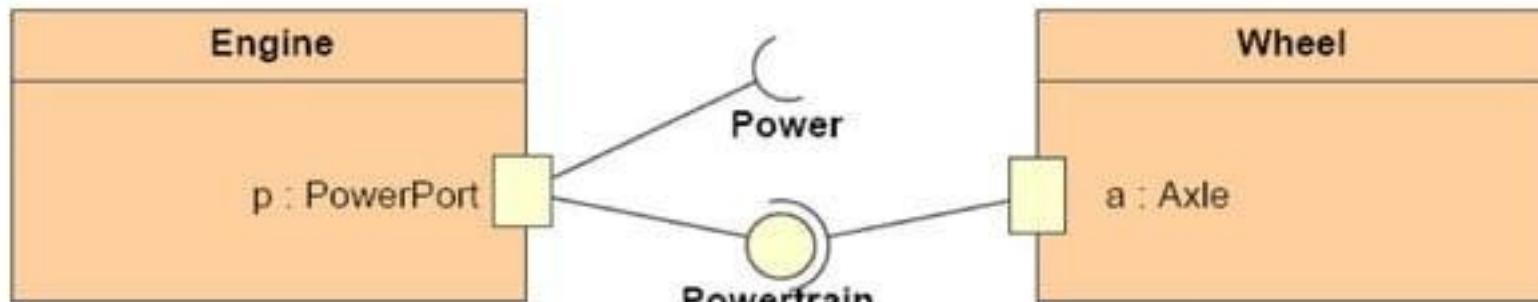
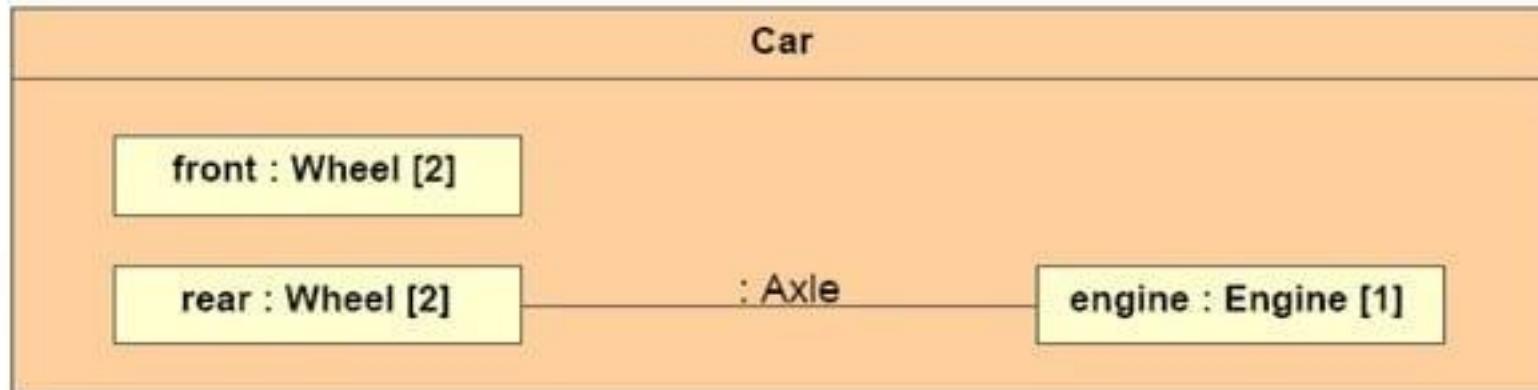
Package Diagram

- Decomposes system into logical units of work
- Describe the dependencies between logical units of work
- Provide views of a system from multiple levels of abstraction



Composite Structure Diagram

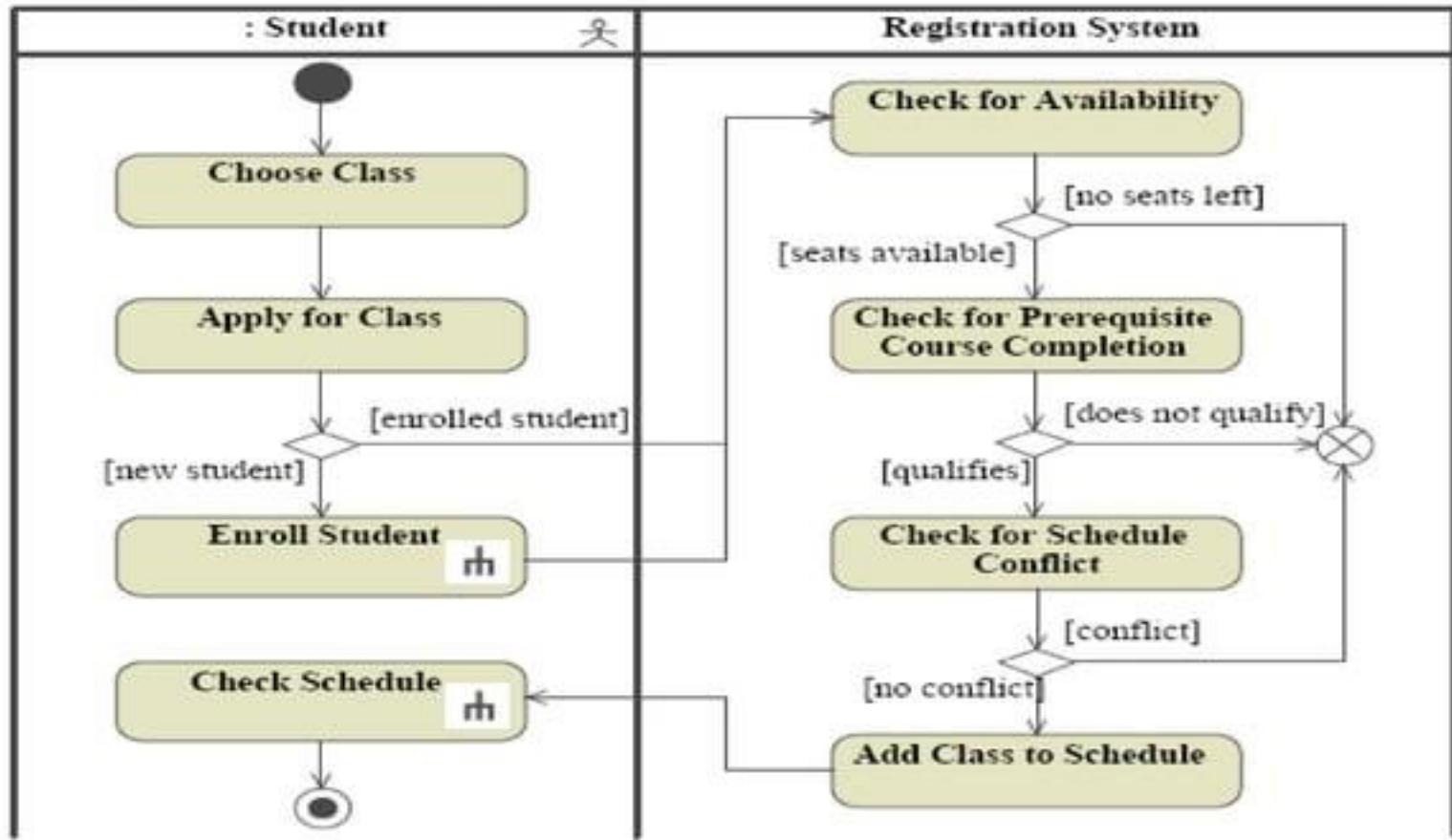
- Shows the internal structure of a classifier, including its interaction points to other parts of the system
- More useful for modeling hardware, real-time systems, integrated device modeling



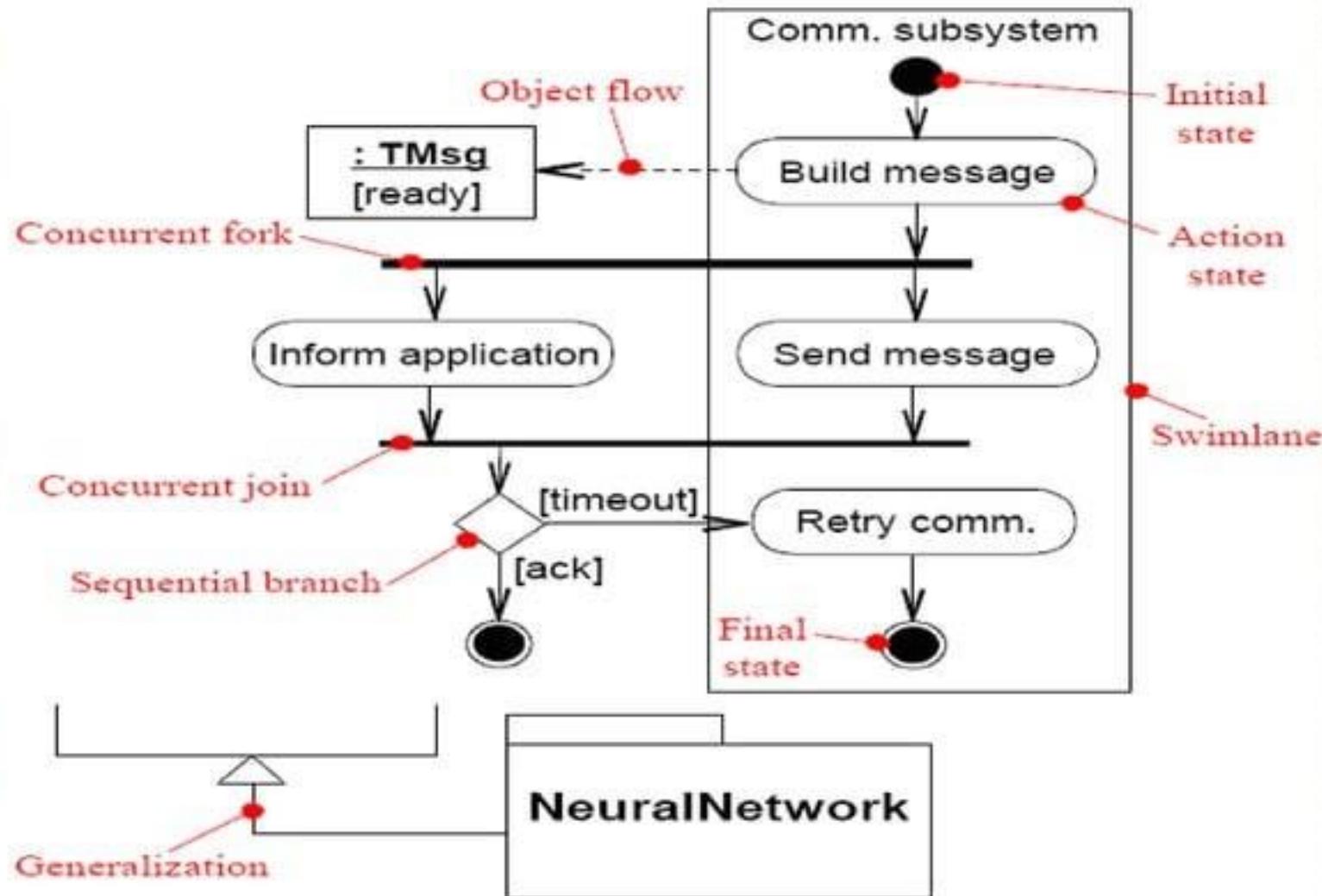
Activity Diagram

Shows a procedural flow for a process

- Useful for workflow modeling
- Supports parallel behavior for multithreaded programming

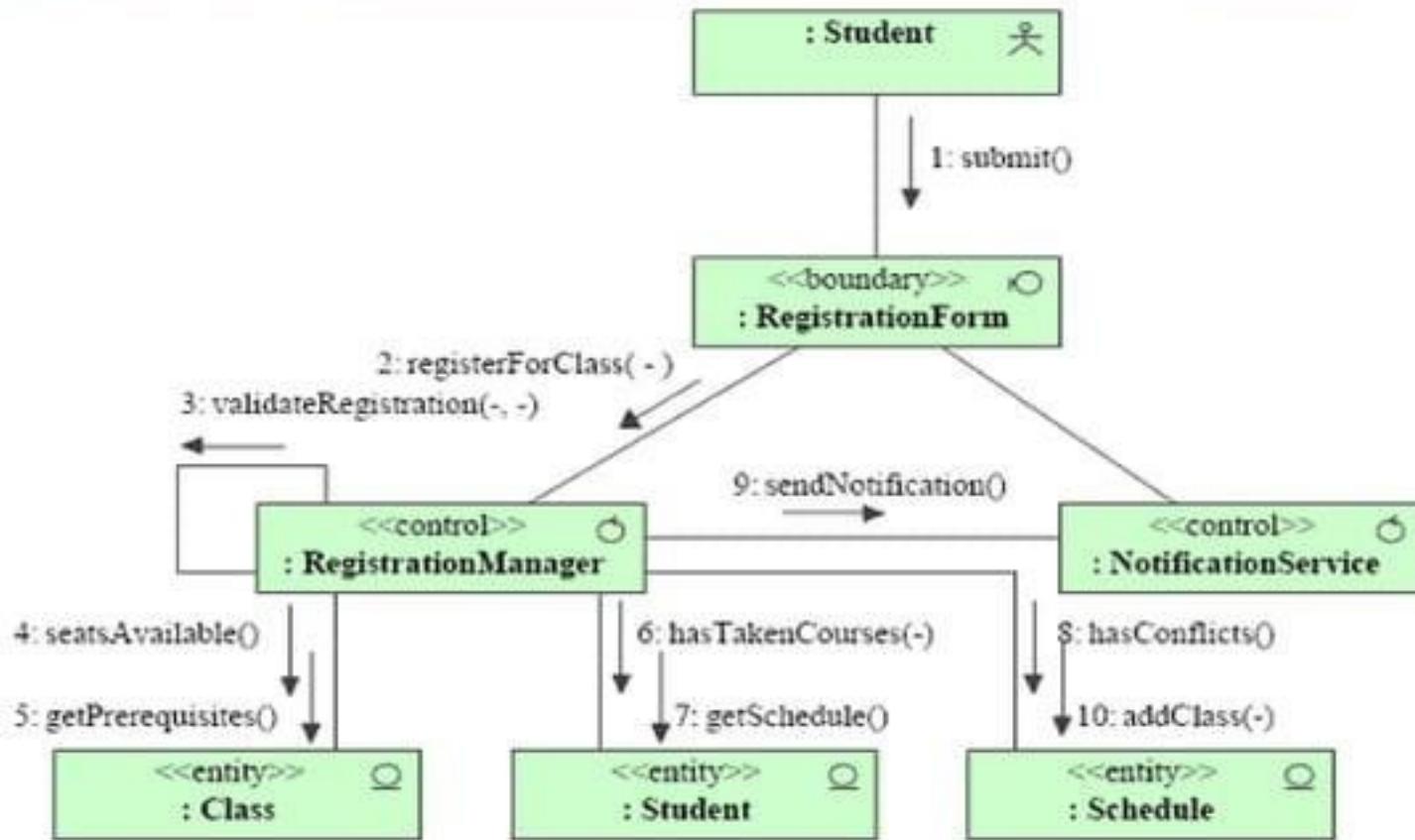


Activity Diagram: example



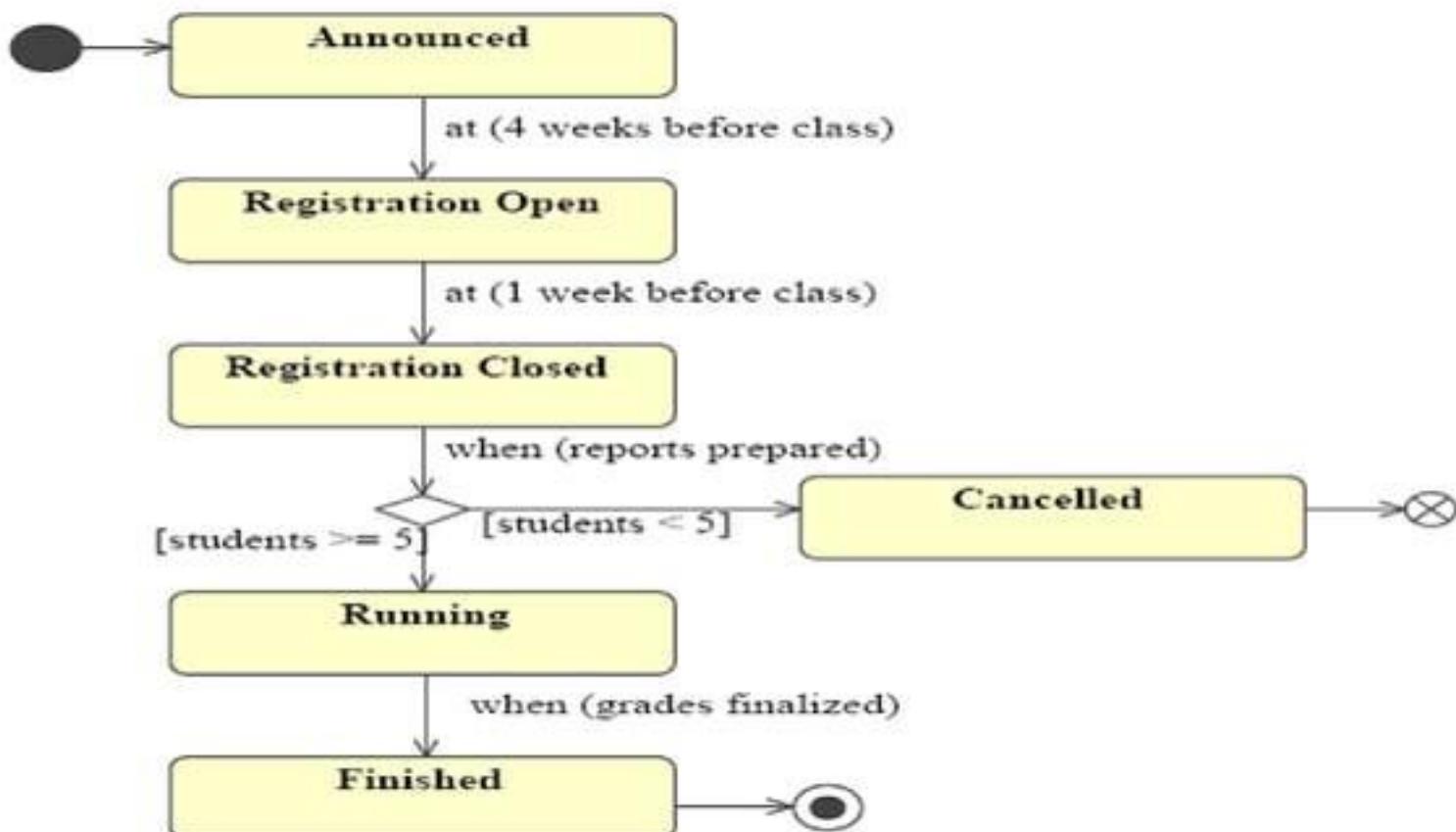
Communication Diagram

- Provides an alternative view to the sequence diagram in a format based on structure rather than time
- Emphasizes how objects interact with each other
- More efficient use of space

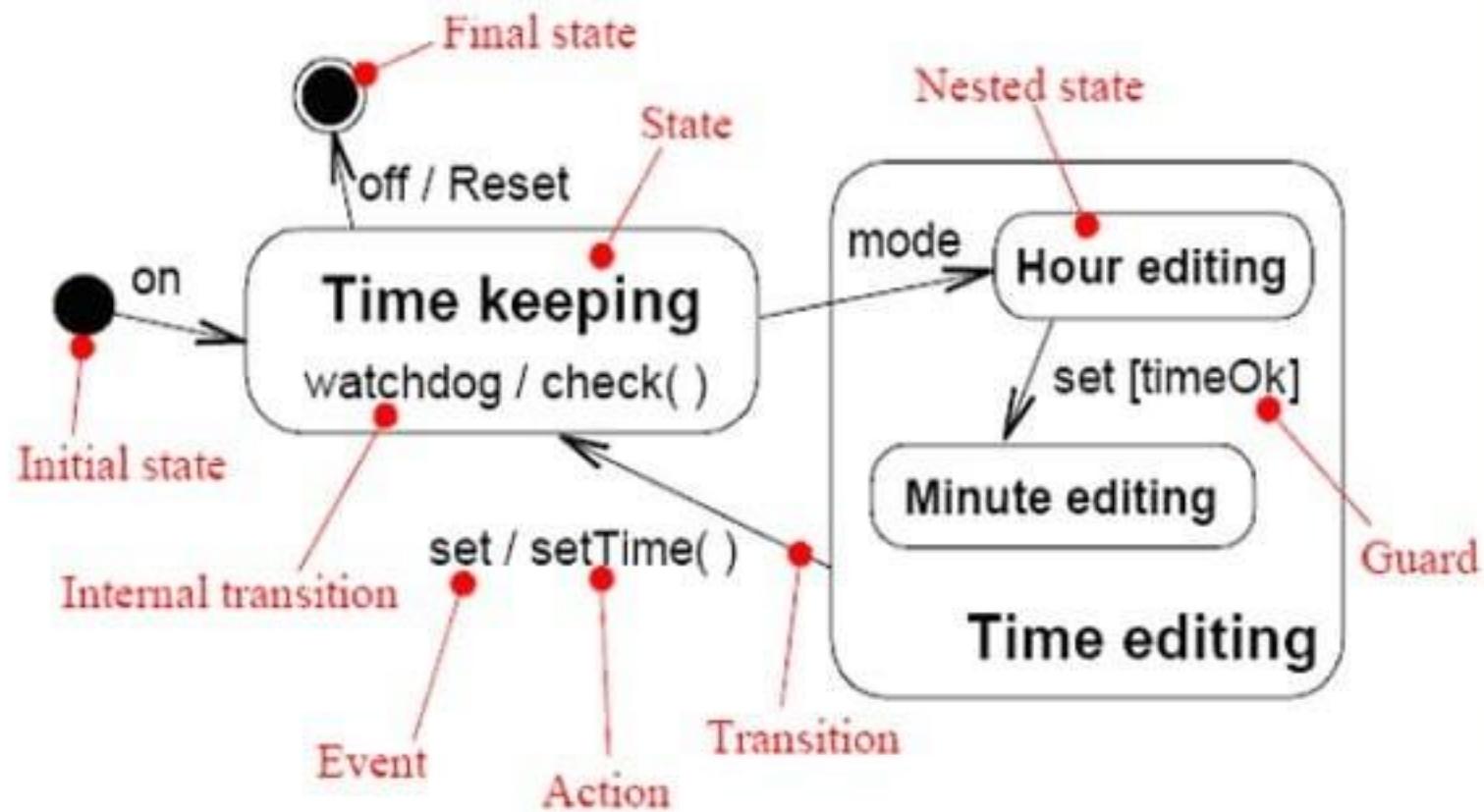


State Diagram

- Describes how an object changes its state that govern its behavior in response to stimuli from the environment
- A statechart diagram is a finite automaton extended with Output (combination of Moore and Mealy automaton)

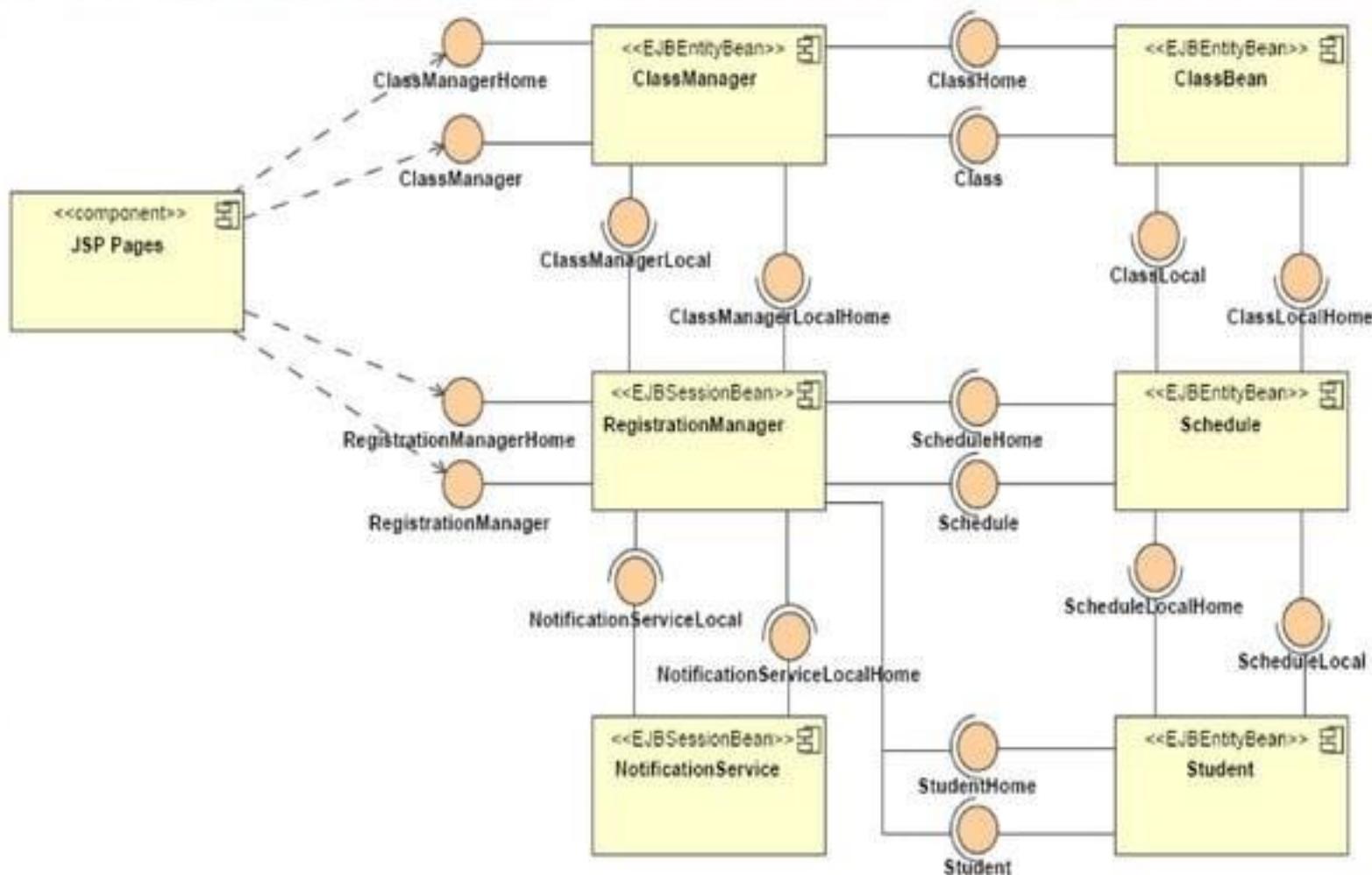


State Diagram:example

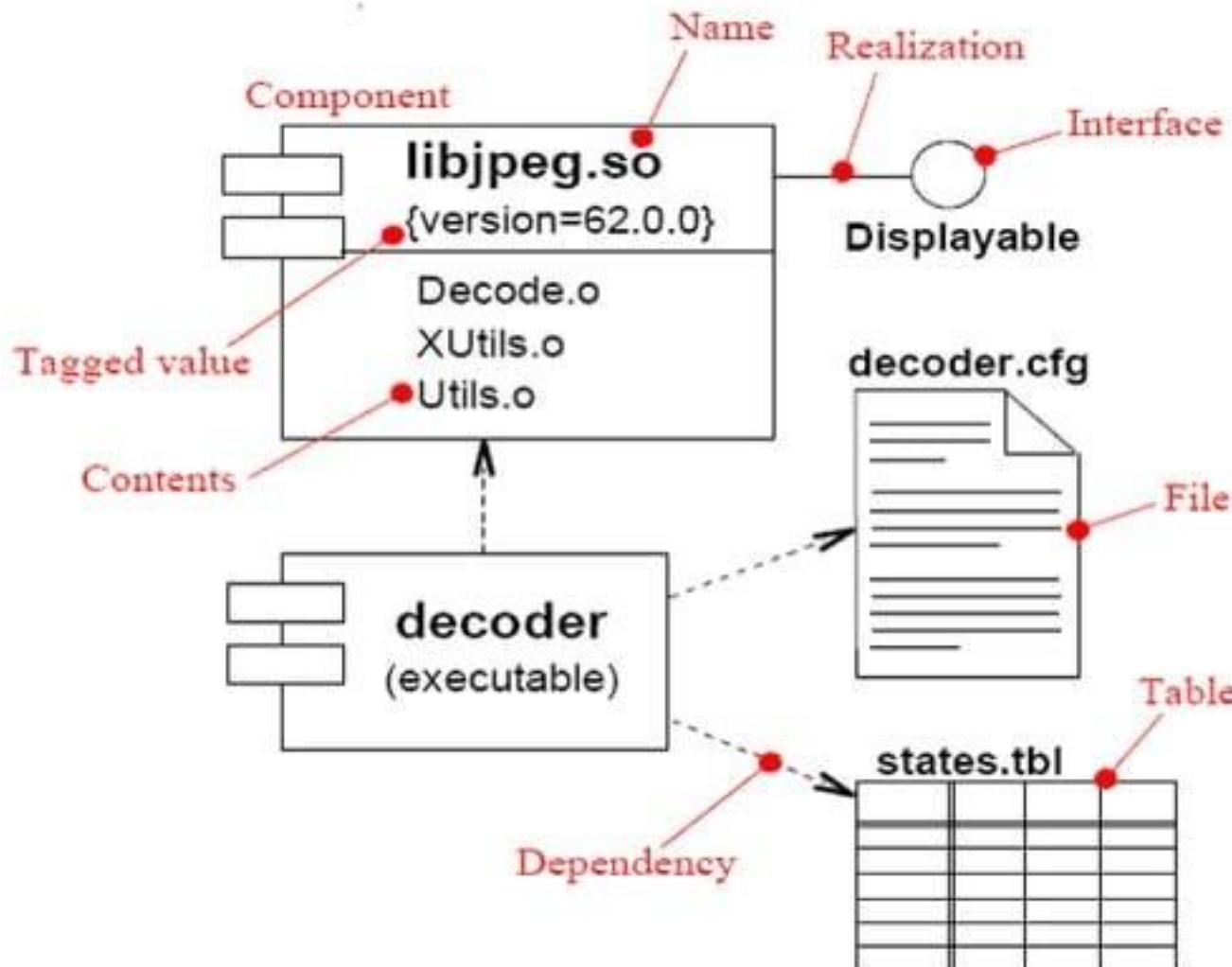


Component Diagram

- Describes software components that make up a system, their interfaces (optional) and relationships

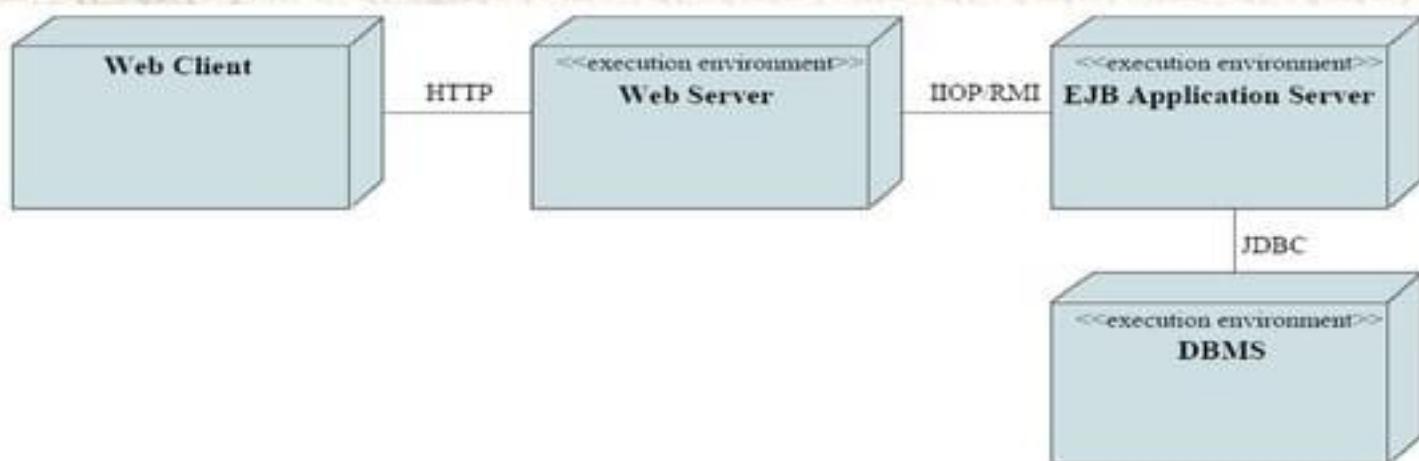


Component Diagram :example



Deployment Diagram

- Describes the configuration of hardware in a system in terms of nodes and connections
- Describes the physical relationships between software and hardware
- Displays how artifacts are installed and move around a distributed system



Reference:

- <http://www.objectsbydesign.com>
 - – UML and OO links, forums, and resources
- <http://www.devx.com/uml/>
 - – UML developer zone
- <http://www.sdmagazine.com/>
 - – Magazine with many UML related articles
- <http://www.omg.org>
 - – The UML Specification and other UML resources

What is a use case diagram?

In the Unified Modeling Language (UML), a use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system. To build one, you'll use a set of specialized symbols and connectors. An effective use case diagram can help your team discuss and represent:

- Scenarios in which your system or application interacts with people, organizations, or external systems
- Goals that your system or application helps those entities (known as actors) achieve
- The scope of your system

When to apply use case diagrams

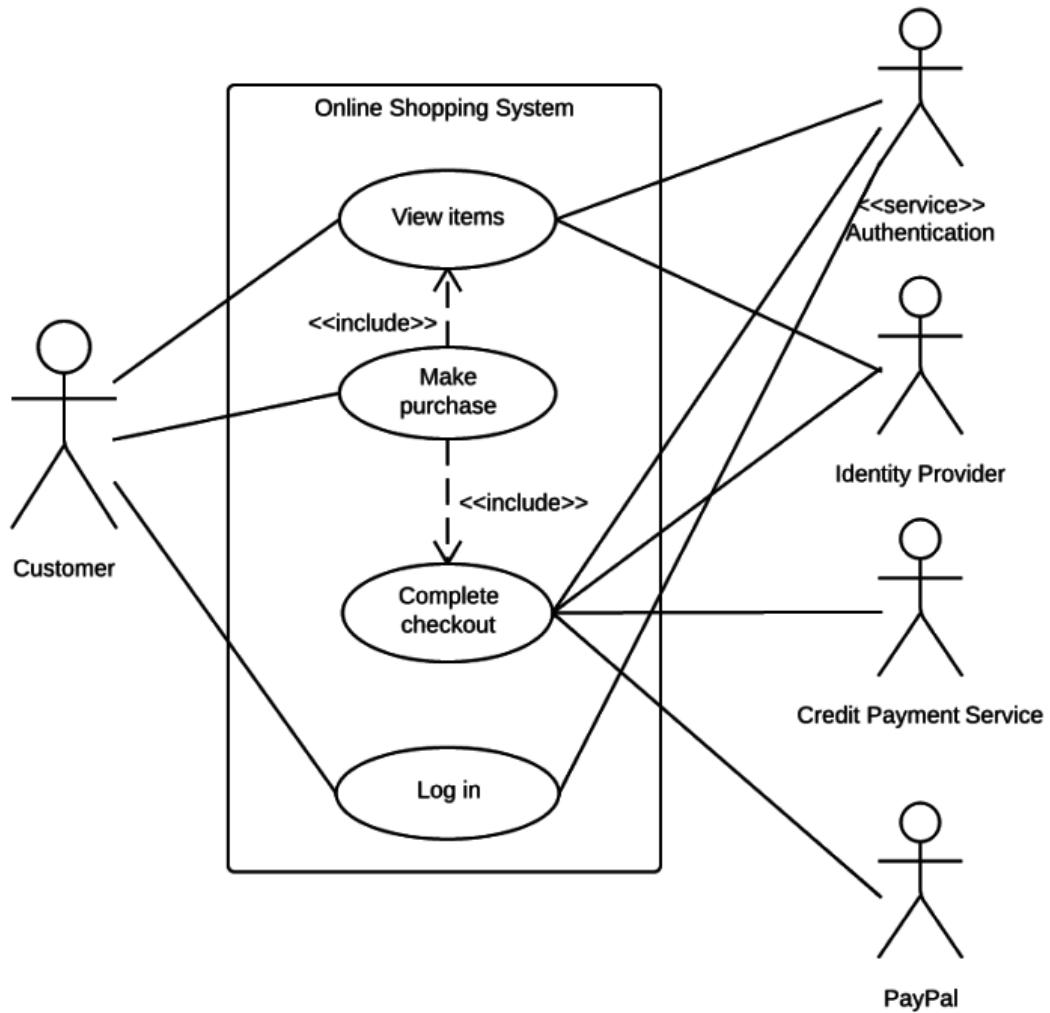
A use case diagram doesn't go into a lot of detail—for example, don't expect it to model the order in which steps are performed. Instead, a proper use case diagram depicts a high-level overview of the relationship between use cases, actors, and systems. Experts recommend that use case diagrams be used to supplement a more descriptive textual use case.

UML is the modeling toolkit that you can use to build your diagrams. Use cases are represented with a labeled oval shape. Stick figures represent actors in the process, and the actor's participation in the system is modeled with a line between the actor and use case. To depict the system boundary, draw a box around the use case itself.

UML use case diagrams are ideal for:

- Representing the goals of system-user interactions
- Defining and organizing functional requirements in a system
- Specifying the context and requirements of a system

- Modeling the basic flow of events in a use case



Diagramming is quick and easy with Lucidchart. Start a free trial today to start creating and collaborating.

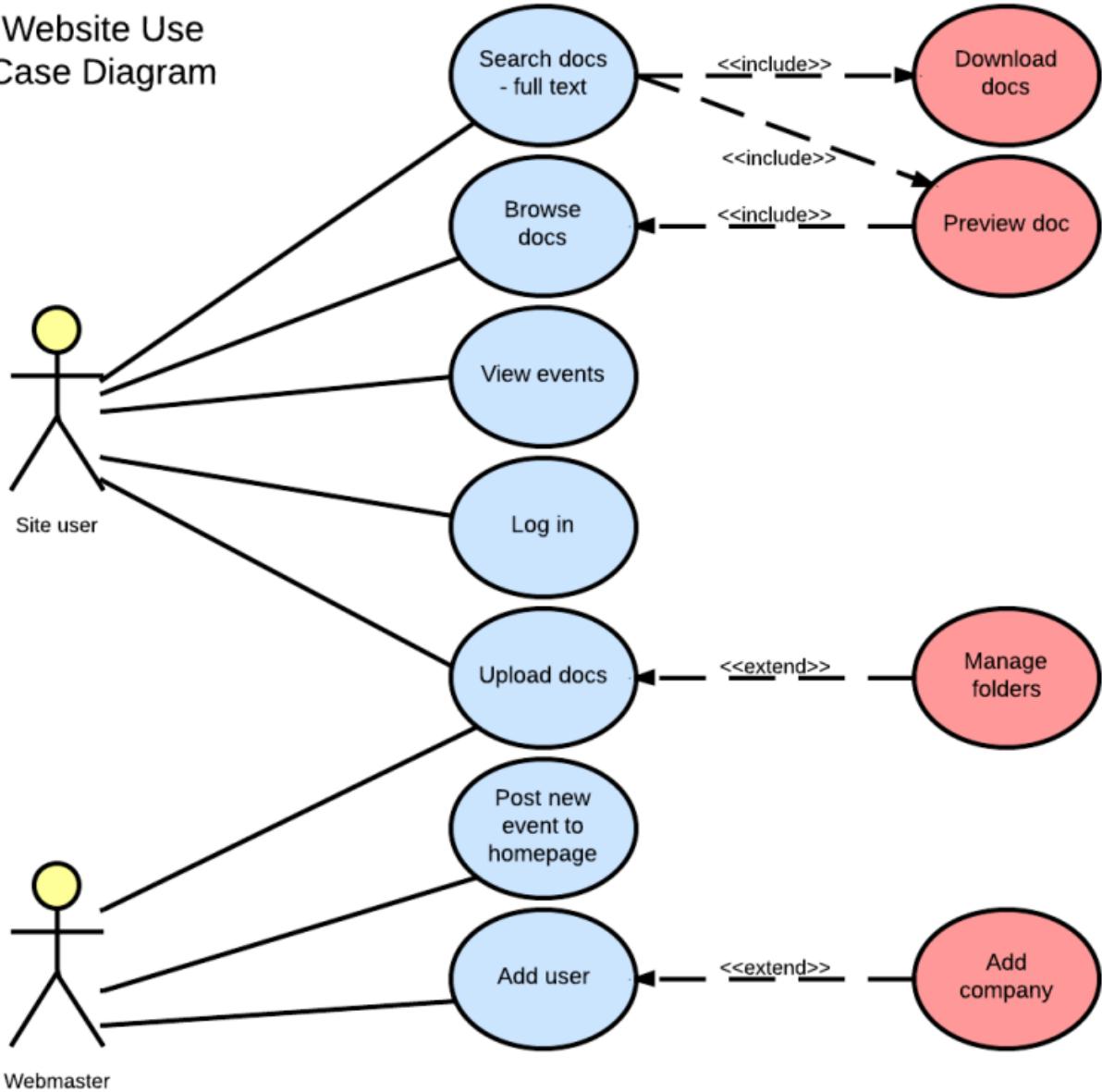
[Create a UML Diagram](#)

Use case diagram components

To answer the question, "What is a use case diagram?" you need to first understand its building blocks. Common components include:

- Actors: The users that interact with a system. An actor can be a person, an organization, or an outside system that interacts with your application or system. They must be external objects that produce or consume data.
- System: A specific sequence of actions and interactions between actors and the system. A system may also be referred to as a scenario.
- Goals: The end result of most use cases. A successful diagram should describe the activities and variants used to reach the goal.

Website Use Case Diagram



Use case diagram symbols and notation

The notation for a use case diagram is pretty straightforward and doesn't involve as many types of symbols as other UML diagrams.

You can use this guide to learn [how to draw a use case diagram](#) if you need a refresher. Here are all the shapes you will be able to find in Lucidchart:

- Use cases: Horizontally shaped ovals that represent the different uses that a user might have.
- Actors: Stick figures that represent the people actually employing the use cases.
- Associations: A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
- System boundary boxes: A box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system. For example, Psycho Killer is outside the scope of occupations in the chainsaw example found below.
- Packages: A UML shape that allows you to put different elements into groups. Just as with component diagrams, these groupings are represented as file folders.

Use case diagram examples

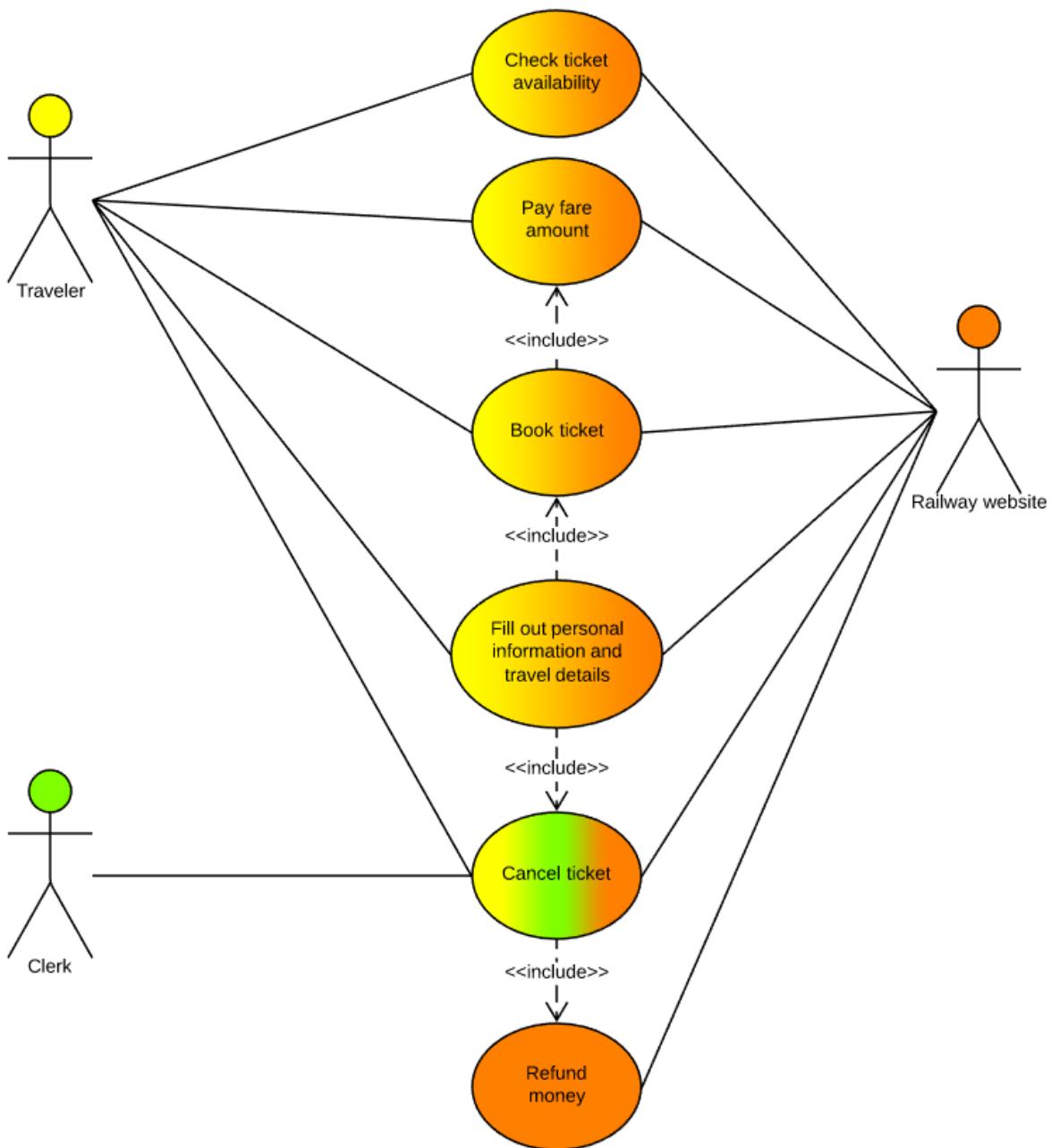
Book publishing use case diagram example

This use case diagram is a visual representation of the process required to write and publish a book. Whether you're an author, an agent, or a bookseller, inserting this diagram into your use case scenario can help your team publish the next big hit. [Try this demo template](#) to get started on your own.



Railway reservation use case diagram example

You can adapt this template for any process where a customer purchases a service. With attractive color schemes, text that's easy to read and edit, and a wide-ranging UML shape library, you're ready to go! Click to [try out this template](#) on your own.



Chainsaw use case diagram example

Consider this example: A man with a chainsaw interacts with the environment around him. Depending on the situation and the context of the situation, he might fall into one of many different use cases. Does he seem to be on his way to work? Is there anything ominous about the way he is wielding his chainsaw? For example, if he is using the chainsaw in a non-occupational setting, we might have reason to think that he falls within the scope of "scary."

