SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

21CSC101T

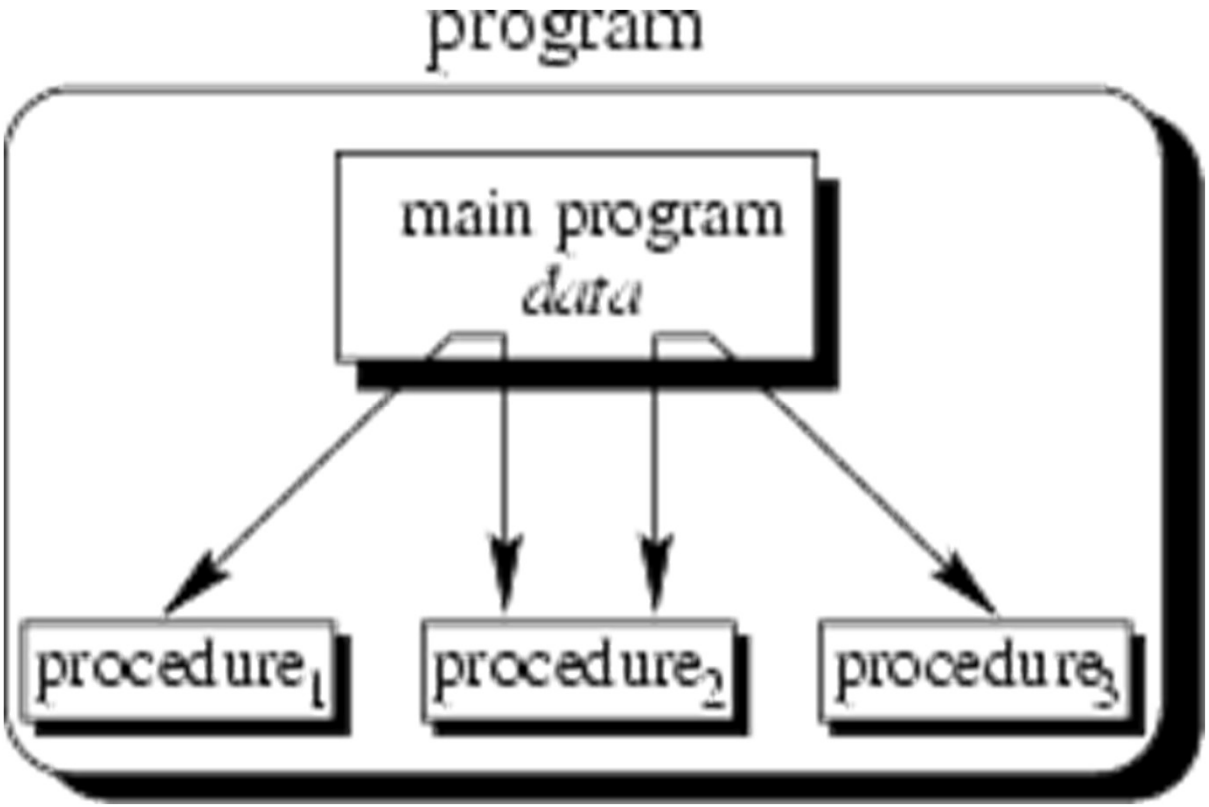
Object Oriented Design and Programming

UNIT-1

21CSC101T - OBJECT ORIENTED DESIGN AND PROGRAMMING

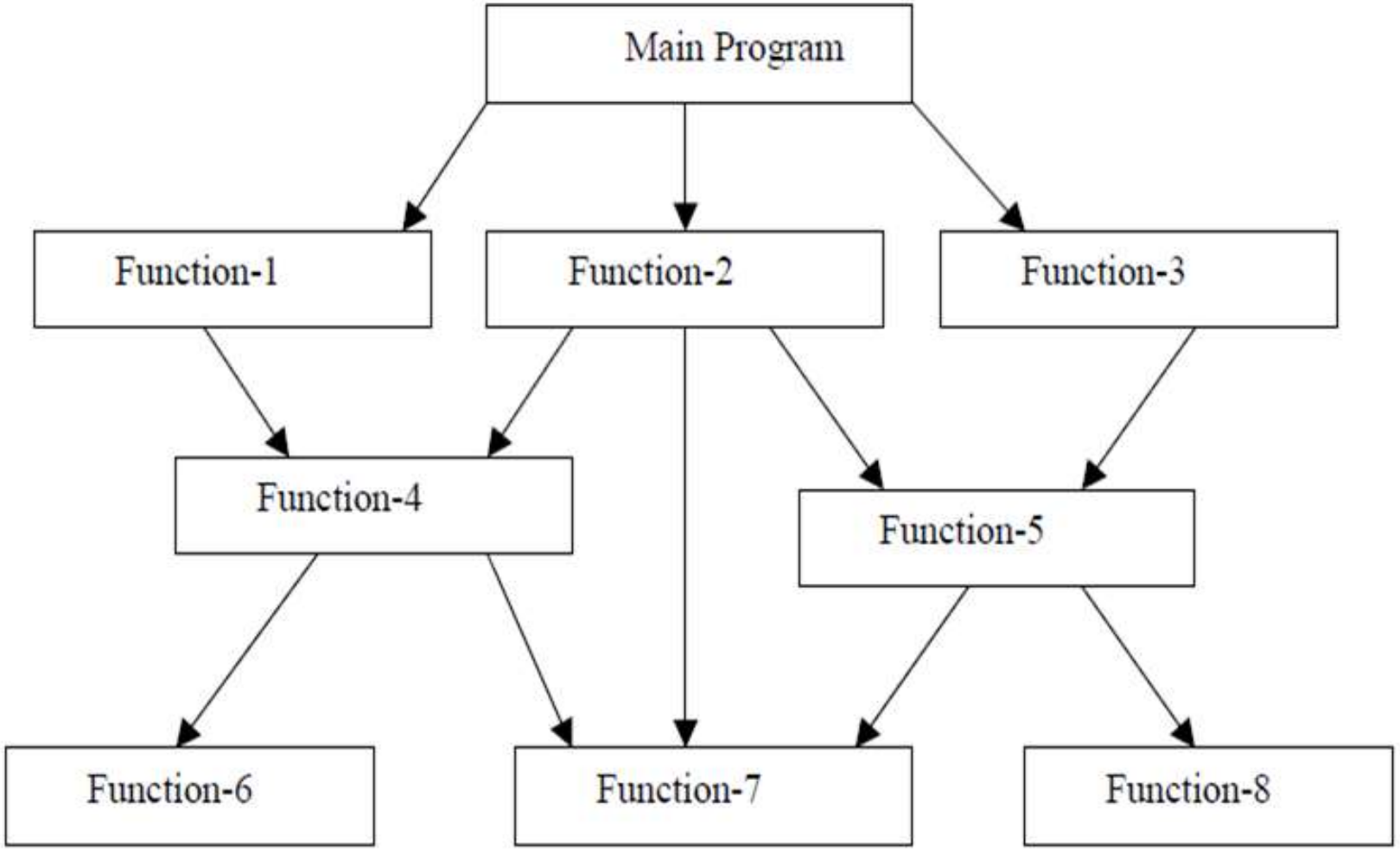
Session 1

Topic : PROCEDURAL AND OBJECT ORIENTED PROGRAMMING



- The main program coordinates calls to procedures and hands over appropriate data as parameters.

Procedure Oriented Programming Language



C function aspects	Syntax
Function declaration	Return-type function-name(argument list); Eg : int add(int a, int b);
Function definition	Return-type function-name(argument list) { body of function;) Eg : int add(int a, int b) {int c; c=a+b; Return c; }
Function call	Function-name(argument list); Eg : add(5,10);

Features of Procedure Oriented Programming Language

- **Smaller programs** - A program in a procedural language is a list of instructions.
- **Larger programs** are divided in to smaller programs known as functions.
- Each **function** has a **clearly defined purpose and a clearly defined interface to the other functions** in the program.
- **Data is Global** and shared by almost all the functions.
- Employs **Top Down approach** in Program Design.

Examples of Procedure Oriented Programming Language

- COBOL
- FORTRAN
- C

Sample COBOL Program(Procedure Oriented)

```
IDENTIFICATION DIVISION.
PROGRAM-ID.

ENVIRONMENT DIVISION.      * procedure will be followed by using few Division

DATA DIVISION.
WORKING-STORAGE SECTION.
77 A PIC 9999.
77 B PIC 9999.
77 ANS PIC 999V99.

PROCEDURE DIVISION.
MAIN-PARA.
                DISPLAY \"--------------------------------\".
                DISPLAY \" ENTER A\"
                ACCEPT A.                  * command line argument
                DISPLAY \"ENTER B\".
                ACCEPT B.
                DISPLAY \"--------------------------------\".

ADD-PARA.
                ADD A B GIVING ANS.
                DISPLAY \"-----------------------------------\".
DISP-PARA.
                DISPLAY \"A IS \" A.
                DISPLAY \"B IS \" B.
                DISPLAY \"ADDITION -\" ANS.
                STOP RUN.                  * to stop the program
```

Disadvantages of Procedural Programming Language

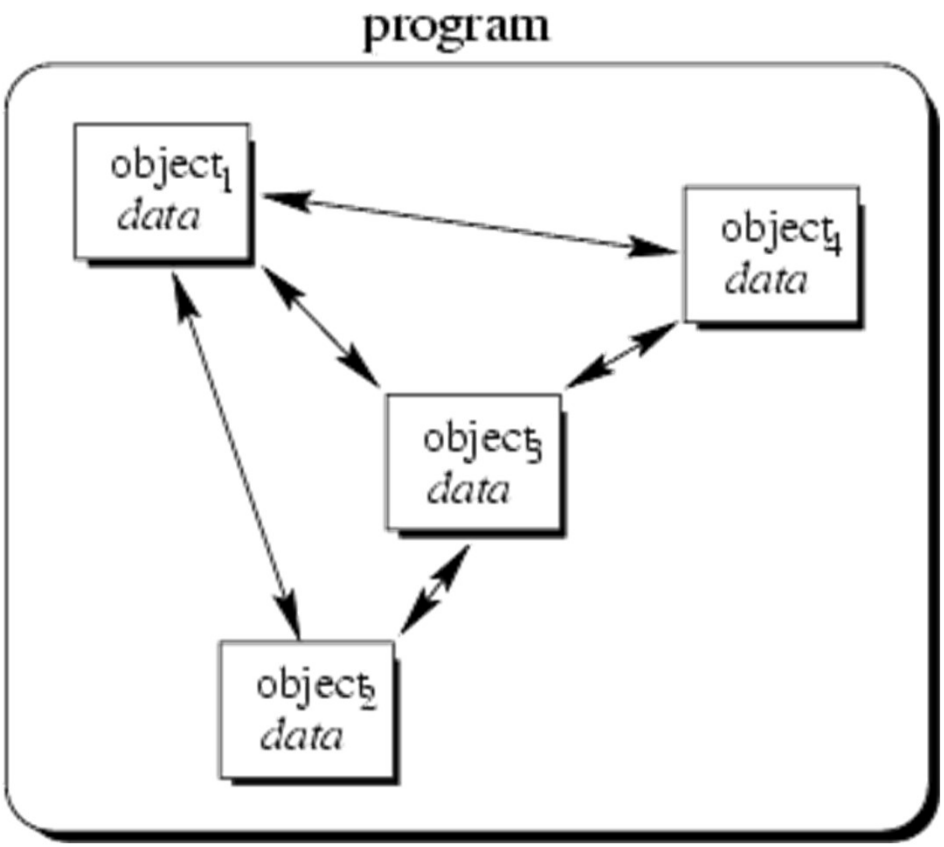
Unrestricted access

- functions have unrestricted access to global data.

Real-world modeling

- unrelated (separated) functions and data, the basis of the procedural paradigm, provide a poor model of the real world.
- Complex real-world objects have both *attributes (data) and behavior (function).*

Object-Oriented Concept



- Objects of the program interact by sending messages to each other, hence it increases the data security (data can be accessed only through its instances)

Object Oriented Programming

- Object-oriented programming is a programming paradigm that uses **abstraction** in the form of **classes and objects** to create models based on the **real world environment**.
- An object-oriented application uses a collection of objects, which communicate by **passing messages** to request services.
- The aim of object-oriented programming is to try to increase the **flexibility and maintainability** of programs.
- Because programs created using an OO language are **modular**, they can be **easier** to develop, and **simpler** to understand after development

Object-Oriented Concepts

- Everything is an object and each object has its **own memory**
- Computation is performed by objects communicating with each other
- Every object is an instance of a class. A class simply represents a grouping of similar objects, such as Integers or lists.
- The class is the repository for behavior associated with an object. That is, that all objects that are instances of the same class can perform the same actions.
- Classes are organized into a singly rooted tree structure, called the **inheritance hierarchy**.
- Memory and behavior associated with instances of a class are automatically available to any class associated with a descendant in this tree structure.

Object-Oriented Programming vs. Procedural Programming

- Programs are made up of modules, which are parts of a program that can be coded and tested separately, and then assembled to form a complete program.
- In procedural languages (i.e. C) these modules are procedures, where a procedure is a sequence of statements.
- The design method used in procedural programming is called Top Down Design.
- This is where you start with a problem (procedure) and then systematically break the problem down into sub problems (sub procedures).

Object-Oriented Programming vs. Procedural Programming

- The difficulties with ***Procedural Programming***, is that software maintenance can be **difficult and time consuming**.
- When changes are made to the main procedure (top), those changes can cascade to the sub procedures of main, and the sub-sub procedures and so on, where the change may impact all procedures in the pyramid.

- Object oriented programming is meant to address the difficulties with procedural programming.

The main difference

Procedural	Object-oriented
Procedure	Method
Record	Object
Module	Class
Procedure call	Message

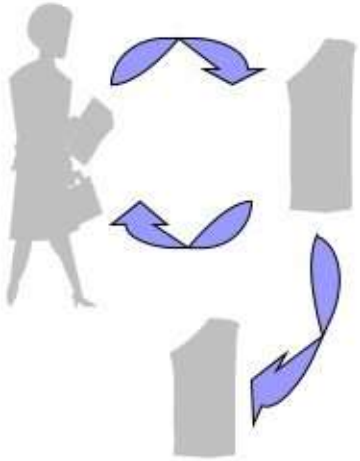
Comparison

Procedural Oriented	Object Oriented
Program is divided into small parts called ‘Functions’	Program is divided into small parts called ‘Objects’
Global and Local data	Has access specifiers : Public, Private, Protected
Doesn’t have any proper way for hiding data	Provides data hiding and security
Eg: C, VB, FORTAN	Eg: C++, JAVA, VB.NET

Comparison

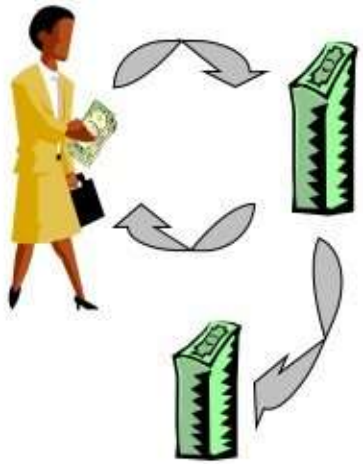
Procedural vs. Object-Oriented

- Procedural



Withdraw, deposit, transfer

- Object Oriented



Customer, money, account

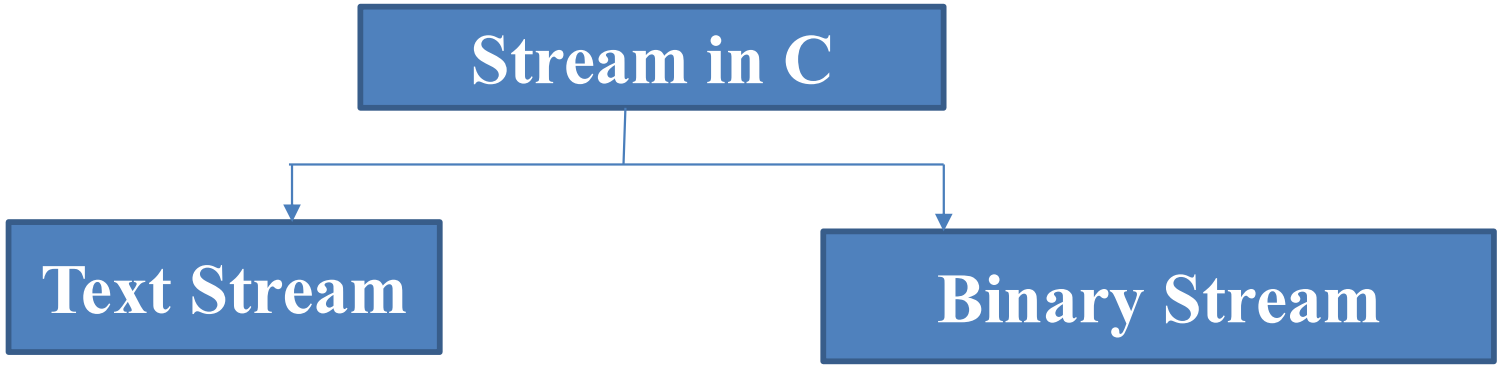
Session-2

Topics covered

- I/O Operation
- Data types
- Variable
- Static
- Constant
- Pointer
- Type conversion

I/O Operation

- C++ I/O operation occurs in streams, which involve transfer of information into byte
- It's a sequences of bytes
- Stream involved in two ways
- It is the source as well as the destination of data
- C++ programs input data and output data from a stream.
- Streams are related with a physical device such as the monitor or with a file stored on the secondary memory.
- In a text stream, the sequence of characters is divided into lines, with each line being terminated.
- With a new-line character (\n) . On the other hand, a binary stream contains data values using their memory representation.


Stream in C
Text Stream
Binary Stream

Cascading of Input or Output Operators

- << operator –It can use multiple times in the same line.
- Its called Cascading
- Cout ,Cin can cascaded

- For example
- cout<<“\n Enter the Marks”;
- cin>> ComputerNetworks>>OODP;

Reading and Writing Characters and Strings

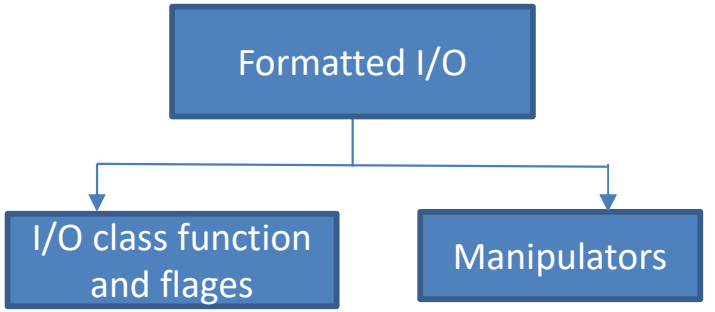
- cin.get(marks);//The value for marks is read
- OR
- marks=cin.get();//A character is read and assigned to marks
- string name;
- Cin>>name;

- string empname;
- cin.getline(empname,20);
- Cout<<“\n Welcome ,”<<empname;

Formatted Input and Output Operations

Table 2.7 Functions for Input/Output

Function	Purpose	Syntax	Usage	Result	Comment
width()	Specifies field size (no. of columns) to display the value	cout. width(w)	cout.width(6);cout<<1239;	1239	It can specify field width for only one value
precision()	Specifies no. of digits to be displayed after the decimal point of a float value	cout. precision(d)	cout. precision(3);cout<<1.234567;	1.234	It retains the setting in effect until it is reset
fill()	Specify a character to fill the unused portion of a field	cout.fill(ch)	Cout.fill('#');cout. width(6);cout<<1239;	##1239	It retains the setting in effect until it is reset




```
#include<iostream.h>
#define PI 3.14159
main()
{
cout.precision(3);
cout.width(10);
cout.fill('*');
cout<<PI;
Output
*****3.142
```

Formatting with flags

- The setf() is a member function of the ios class that is used to set flags for formatting output.
- **syntax -cout.setf(flag, bit-field)**
- Here, flag defined in the ios class specifies how the output should be formatted
- bit-field is a constant (defined in ios) that identifies the group to which the formatting flag belongs to.
- There are two types of setf()—one that takes both flag and bit-fields and the other that takes only the flag .

Table 2.8 Types of setf()

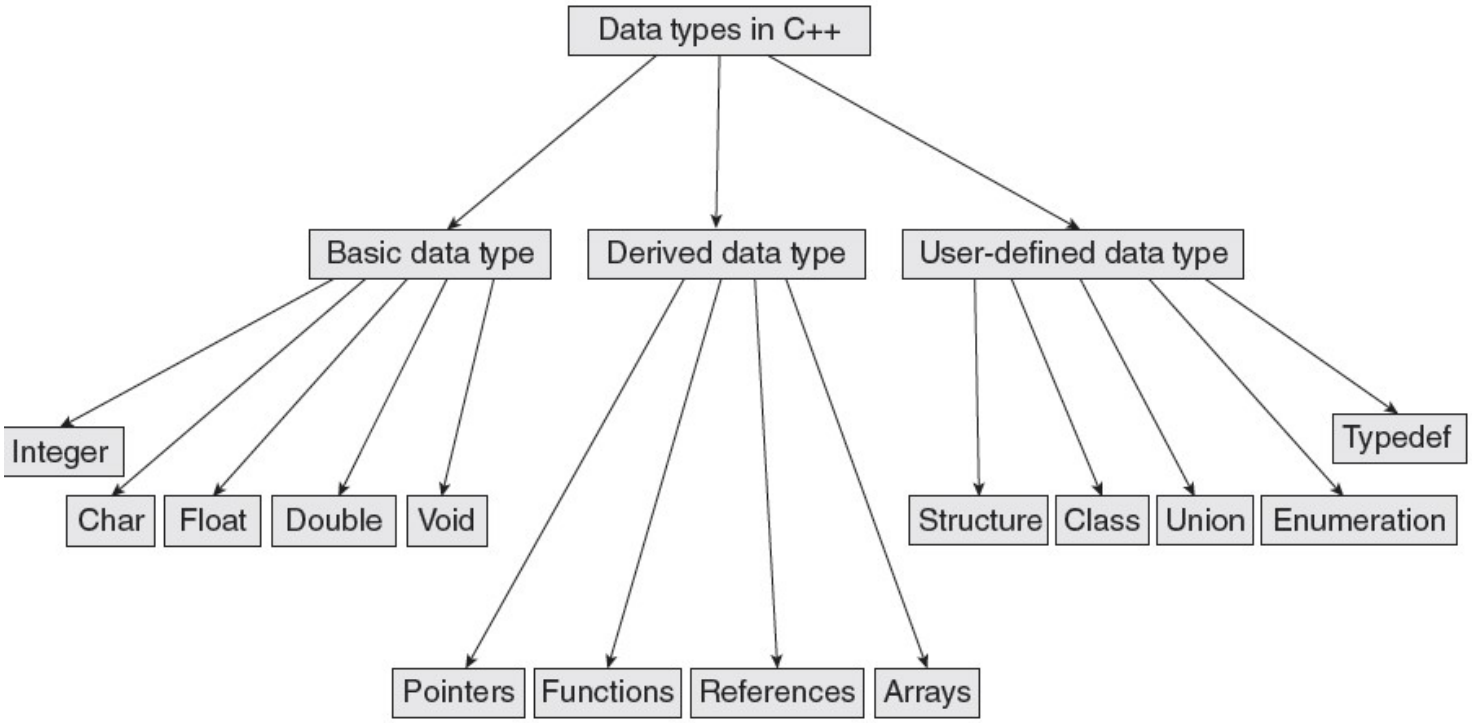
Flag	Bit-field	Usage	Purpose	Comment
ios :: left	ios :: adjustifield	cout.setf(ios :: left, ios :: adjustifield)	For left justified output	If no flag is set, then by default, the output will be right justified.
ios :: right	ios:: adjustifield	cout.setf(ios :: right, ios :: adjustifield)	For right justified output	
los :: internal	ios:: adjustifield	cout.setf(ios :: internal, ios :: adjustifield)	Left-justify sign or base indicator, and right-justify rest of number	
ios :: scientific	ios:: floatifield	cout.setf(ios :: scientific, ios :: floatifield)	For scientific notation	If no flag is set then any of the two notations can be used for floating point numbers depending on the decimal point
ios :: fixed	ios :: floatifield	cout.setf(ios :: fixed, ios :: floatifield)	For fixed point notation	
ios :: dec	ios :: basefield	cout.setf(ios :: dec, ios :: basefield)	Displays integer in decimal	If no flag is set, then the output will be in decimal.
ios :: oct	ios :: basefield	cout.setf(ios :: oct, ios :: basefield)	Displays integer in octal	
ios :: hex	ios :: basefield	cout.setf(ios :: hex, ios :: basefield)	Displays integer in hexadecimal	
ios :: showbase	Not applicable	cout.setf(ios :: showbase)	Show base when displaying integers	
ios :: showpoint	Not applicable	cout.setf(ios :: showpoint)	Show decimal point when displaying floating point numbers	
ios :: showpos	Not applicable	cout.setf(ios :: showpos)	Show + sign when displaying positive integers	
ios :: uppercase	Not applicable	cout.setf(ios :: uppercase)	Use uppercase letter when displaying hexadecimal (OX) or exponential numbers (E)	

Formatting Output Using Manipulators

certain manipulators to format the output

Manipulator	Purpose	Usage	Result	Alternative
setw(w)	Specifies field size (number of columns) to display the value	cout<<setw(6) <<1239;	␣ ␣ 1 2 3 9	width()
setprecision(d)	Specifies number of digits to be displayed after the decimal point of a float value	cout<<setprecision(3) <<1.234567;	1.235	precision()
setfill(c)	Specify a character to fill the unused portion of a field	cout<<setfill('#') <<setwidth(6) <<1239;	# # 1 2 3 9	fill()

Data Types in C++



Data type	Size in bytes	Range
Char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
Int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed short int	2	-32768 to 32767
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
long int	4	-2147483648 to 214748364
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 214748364
Float	4	3.4E-38 to 3.4E+38
Double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Variables

A variable is the content of a memory location that stores a certain value. A variable is identified or denoted by a variable name. The variable name is a sequence of one or more letters, digits or underscore, for example: character_

Rules for defining variable name:

- A variable name can have one or more letters or digits or underscore for example character_.
- White space, punctuation symbols or other characters are not permitted to denote variable name.
- A variable name must begin with a letter.
- Variable names cannot be keywords or any reserved words of the C++ programming language.
- Data C++ is a case-sensitive language. Variable names written in capital letters differ from variable names with the same name but written in small letters.

01 Local Variables

02 Static Variables

03 Instance variables

04 Final Variables

Variables

Local Variables

Local variable: These are the variables which are declared within the method of a class.

Example:

```
public class Car {
public:
 void display(int m)
  {
        int model=m;
        cout<<model;
   }
```

Instance Variables

Instance variable: These are the variables which are declared in a class but outside a method, constructor or any block.

Example:

```
public class Car {
     private:
String color;
 Car(String c)
  {
   color=c;
  }}
```

Static Variables

Static variables: Static variables are also called as class variables. These variables have only one copy that is shared by all the different objects in a class.

Example:

```
public class Car {
     public:
 static int tyres;
void init()
 {
        tyres=4;
} }
```

Constant Variables

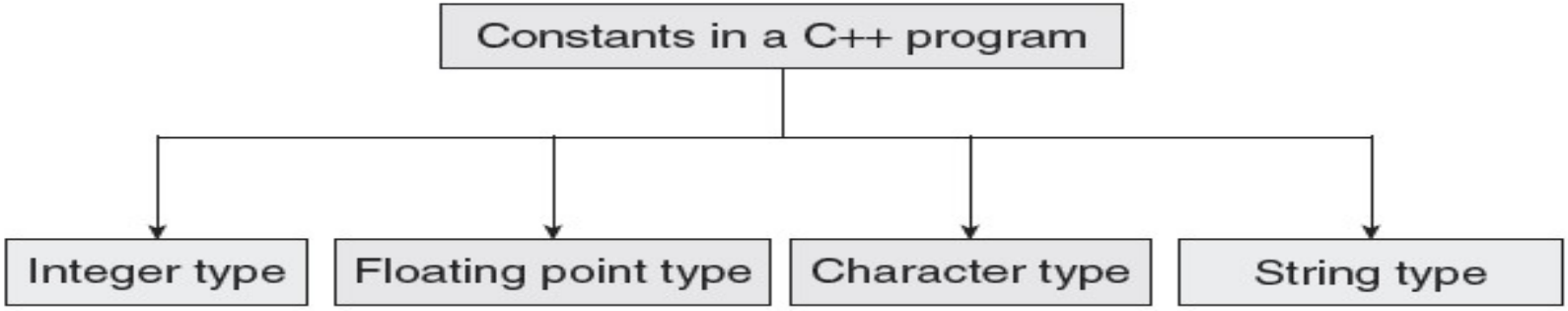
Constant is something that doesn't change. In C language and C++ we use the keyword const to make program elements constant.

Example:

```
const int i = 10;
void f(const int i)
class Test
{
   const int i;
};
```

CONSTANTS

- Constants are identifiers whose value does not change. While variables can change their value at any time, constants can never change their value.
- Constants are used to define fixed values such as Pi or the charge on an electron so that their value does not get changed in the program even by mistake.
- A constant is an explicit data value specified by the programmer.
- The value of the constant is known to the compiler at the compile time.



Declaring Constants

- Rule 1 Constant names are usually written in capital letters to visually distinguish them from other variable names which are normally written in lower case characters.
- Rule 2 No blank spaces are permitted in between the # symbol and define keyword.
- Rule 3 Blank space must be used between #define and constant name and between constant name and constant value.
- Rule 4 #define is a preprocessor compiler directive and not a
  l with a semi-colon.

```
const data_type const_name = value;
```

The const keyword specifies that the value of pi cannot change.

```
const float pi = 3.14;
```

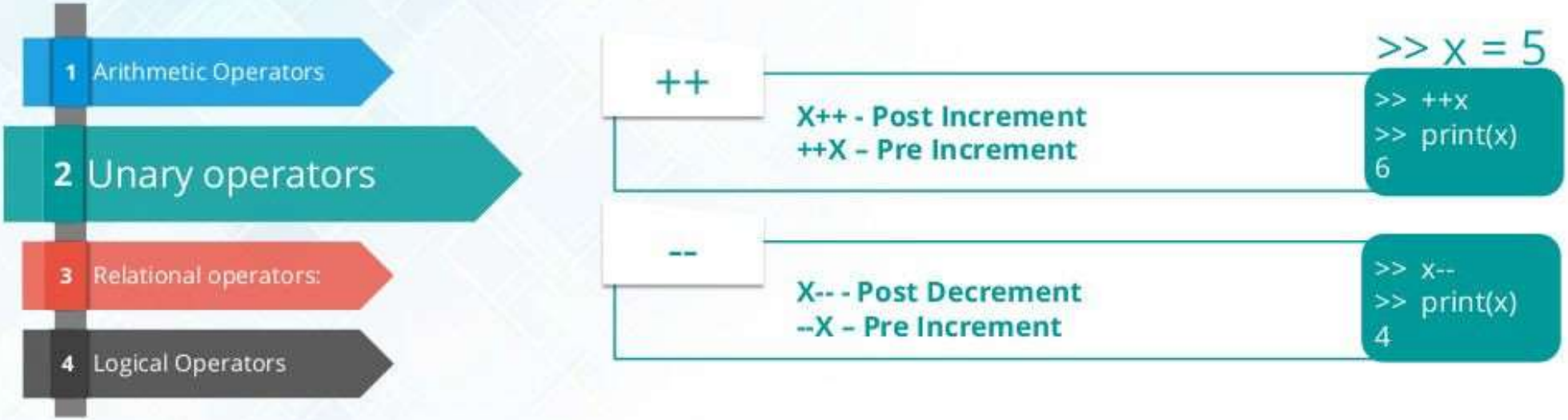
```
#define PI 3.14159
#define service_tax 0.12
```

Data Operators

Arithmetic Operators

+	Add two operands or unary plus	>> 2 + 3 5 >> +2
-	Subtract two operands or unary subtract	>> 3 – 1 2 >> -2
*	Multiply two operands	>> 2 * 3 6
/	Divide left operand with the right and result is in float	>> 6 / 3 2.0
%	Remainder of the division of left operand by the right	>> 5 % 3 2

Unary operators



Relational operators

>	True if left operand is greater than the right	>> 2 > 3 False
<	True if left operand is less than the right	>> 2 < 3 True
==	True if left operand is equal to right	>>2 == 2 True
!=	True if left operand is not equal to the right	>> x >>= 2 >>print(x) 1

Logical operators

1 Arithmetic Operators

2 Unary operators

3 Relational operators:

4 Logical Operators

and

Returns True if both x and y are True, False otherwise

```
>> True
&& False
False
```

or

Returns True if at least x or y are True, False otherwise

```
>> True ||
False
True
```

not

Returns True if x is False, True otherwise

```
>> ! 1
False
```

Special Operators

1

Scope resolution operator

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by using scope resolution operator (::), because this operator allows access to the global version of a variable.

2

New Operator

The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

3

Delete Operator

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

4

Member Operator

C++ permits us to define a class containing various types of data & functions as members. To access a member using a pointer in the object & a pointer to the member.

Operator Precedence

Rank	Operators	Associativity
1	++, --, +, -, !(logical complement) ,~(bitwise complement) ,(cast)	Right
2	*(mul) , /(div), %(mod)	Left
3	+ , - (Binary)	Left
4	>> , <<, >>> (Shift operator)	Left
5	>, <, >=,<=	Left
6	==, !=	Left
7	& (Bitwise and)	Left
8	^ (XOR)	Left
9	| (Bitwise OR)	Left
10	&& (Logical and)	Left
11	|| (Logical OR)	Left
12	Conditional	Right
13	Shorthand Assignment	Right

POINTERS

Introduction

- Pointers are symbolic representation of addresses.
- They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures.
- Syntax data_type *pointer_variable;
- Example

  int *p,sum;

Assignment

- integer type pointer can hold the address of another int variable
- To assign the address of variable to pointer-**ampersand symbol** (&)
- p=∑

this pointer

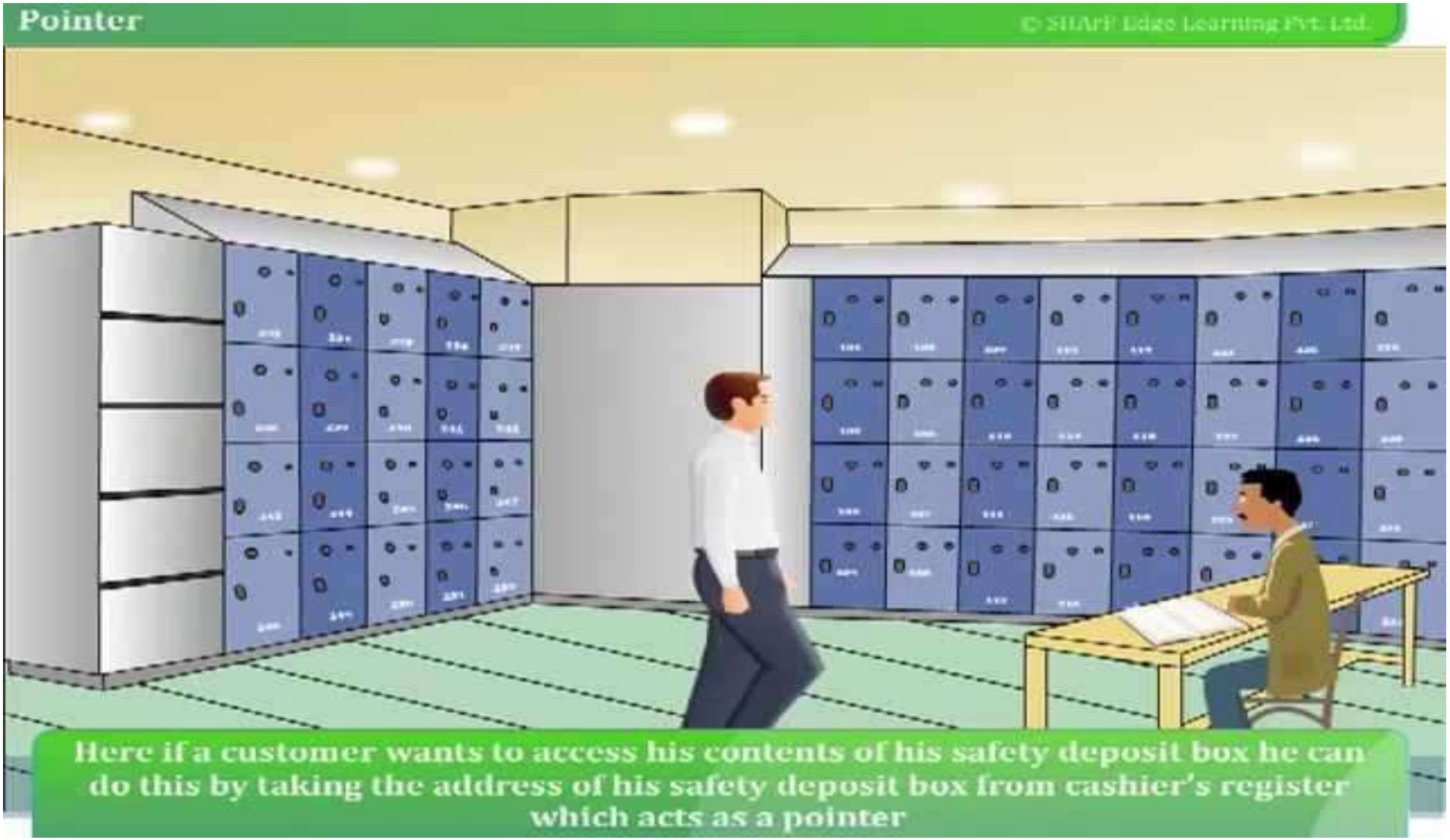
hold the adderss of current object

- int num;
- This->num=num;

Void?

- **When used in the declaration of a pointer, void specifies that the pointer is "universal."** If a pointer's type is void* , the pointer can point to any variable that's not declared with the const or volatile keyword.

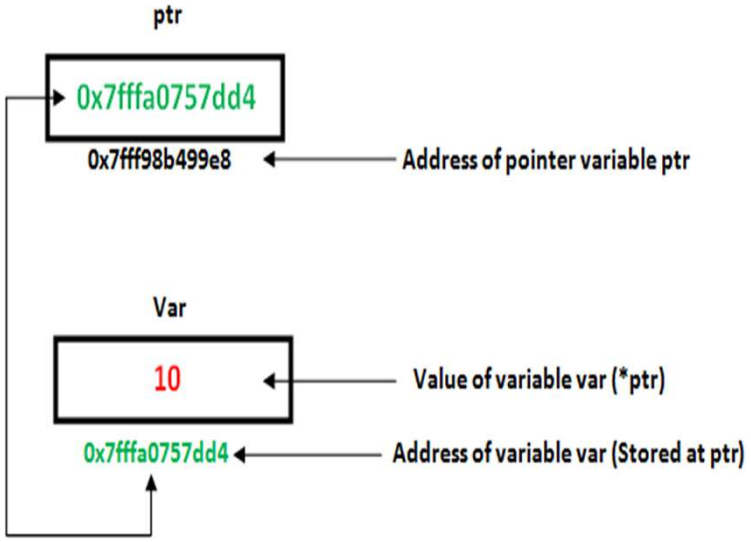
Real Time Example



How to use it

- P=∑//assign address of another variable
- cout<<∑ //to print the address of variable
- cout<<p;//print the value of variable
- Example of pointer

```
#include<iostream.h>
using namespace std;
int main()
{ int *p,sum=10;
p=&sum;
cout<<“Address of sum:”<<&sum<<endl;
cout<<“Address of sum:”<<p<<endl;
cou<<“Address of p:”<<&p<<endl;
cout<<“Value of sum”<<*p;
}
```



Output:
Address of sum : 0X77712
Address of sum: 0x77712
Address of p: 0x77717
Value of sum: 10

Pointers and Arrays

- assigning the address of array to pointer don't use ampersand sign(&)

```
#include <iostream>
using namespace std;
int main(){
  //Pointer declaration
  int *p;
  //Array declaration
  int arr[]={1, 2, 3, 4, 5, 6};
  //Assignment
  p = arr;
  for(int i=0; i<6;i++){
    cout<<*p<<endl;
    //++ moves the pointer to next int position
    p++;
  }
  return 0;
}
```

OUTPUT:

1
2
3
4
5
6

This Pointers

- this pointer hold the address of current object
- int num;
- This->num=num;

This pointer Example

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id; //data member (also instance variable)
     string name; //data member(also instance variable)
     float salary;
     Employee(int id, string name, float salary)
      {
          this->id = id;
         this->name = name;
         this->salary = salary;
      }
     void display() {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      } };
int main(void) {
Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
   e1.display();    e2.display();      return 0;}
```

Output: 101 Sonoo 890000

102 Nakul 59000

Function using pointers

```
#include<iostream>
using namespace std;
void swap(int *a ,int *b );
//Call By Reference
int main()
{
  int p,q;
 cout<<"\nEnter Two Number You Want To Swap
\n";
cin>>p>>q;
swap(&p,&q);
cout<<"\nAfter Swapping Numbers Are Given
below\n\n";
cout<<p<<"   "<<q<<" \n";
    return 0;
}
```

```
void swap(int *a,int *b)
{
int c;
c=*a;
*a=*b;
*b=c;
}
```

Output:

```
Enter Two Number You Want to
Swap
10 20
After Swapping Numbers Are
Given below
20 10
```

Type conversion and type casting

- **Type conversion** or typecasting of variables refers to changing a variable of one data type into another.
- While type conversion is done implicitly, casting has to be done explicitly by the programmer.

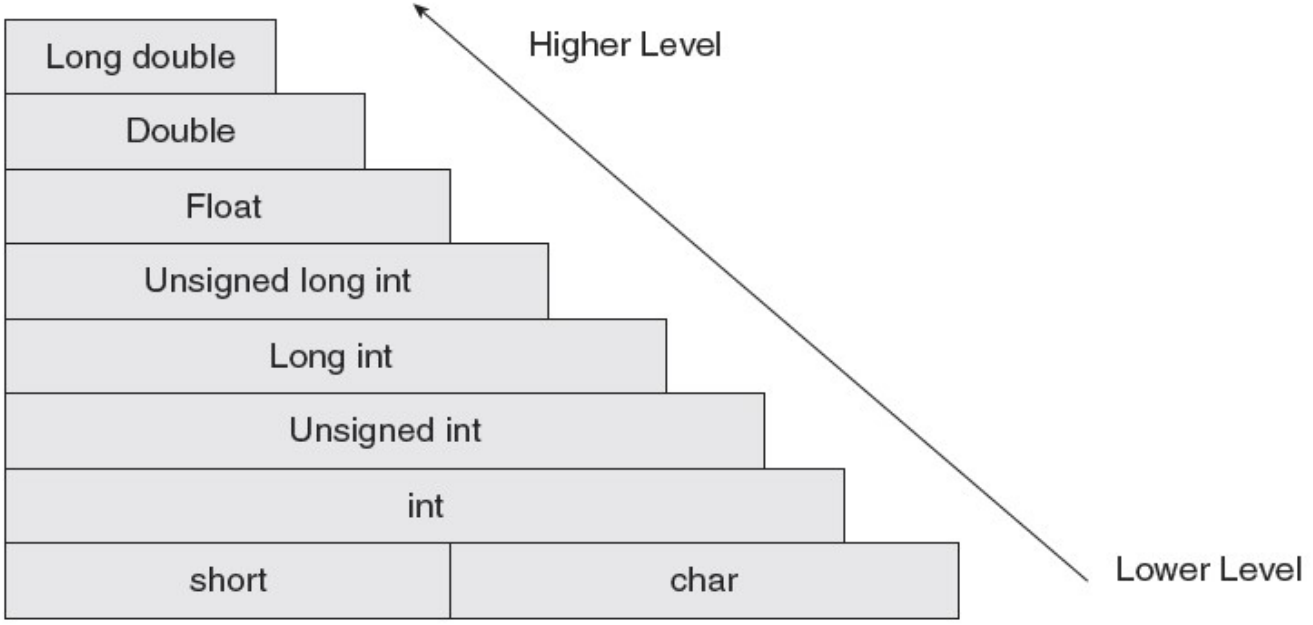


Figure 2.13 Conversion hierarchy of data types

Implicit Type Conversion

The type conversion that is done automatically done by the compiler is known as implicit type conversion. This type of conversion is also known as automatic conversion.

```
#include <iostream>
using namespace std;
int main()
{
int x = 10;
// integer x
char y = 'a';
// character y implicitly converted to
int. ASCII
// value of 'a' is 97
x = x + y;
// x is implicitly converted to float
int z = x + 1;
cout << "x = " << x << endl<< "y = " << y
<< endl<< "z = " << z << endl;
return 0;}
```

Output:

```
x = 107
y = a
z = 108
```

Type Casting

- Type casting an arithmetic expression tells the compiler to represent the value of the expression in a certain format.
- It is done when the value of a higher data type has to be converted into the value of a lower data type.
- However, this cast is under the programmer's control and not under the compiler's control.
- Syntax:
- destination_variable_name=destination_data_type(source_variable_name);

```
float sal=10000.00;
int income;
Income=int(sal);
```

Example for Explicit Type Conversion

```
#include <iostream>
using namespace std;
int main()
{    double x = 1.2;
// Explicit conversion from double to int
int sum = (int)x + 1;
cout << "Sum = " << sum;
return 0;
}
```

Output:

Sum = 2

Session-3
Class and objects - Features

Class

- Class is a user defined data type, defined using keywork **class.**
- It holds its own data members and member functions,
- It can be accessed and used by creating instance of that class.
- The variables inside class definition are called as data members and

  the functions are called member functions

```
class className
{
   data members
   member functions
};
```

- Class is just a blue print, which declares and defines characteristics and behavior, namely data members and member functions respectively.
- All objects of this class will share these characteristics and behavior.
- Class name must start with an uppercase letter(Although this is not mandatory).

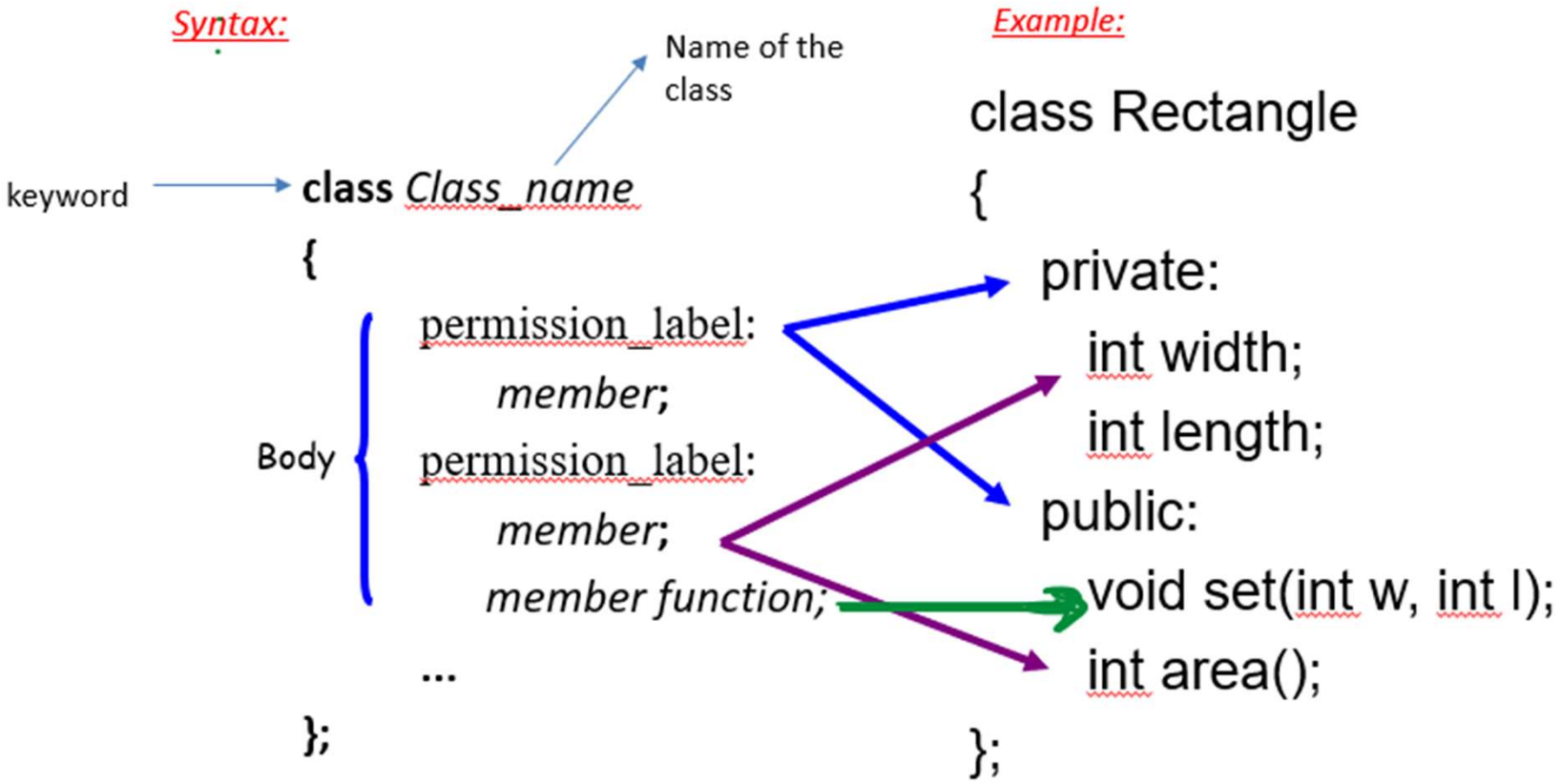
Example,

class Student

Example

```
class Room {
   public:
      double length;
      double breadth;
      double height;

      double calculateArea(){
         return length * breadth;
      }

      double calculateVolume(){
         return length * breadth *
height;
      }

};
```

Define a Class Type



- Data members Can be of any type, built-in or user-defined.

This may be,

- non-static data member

  Each class object has its own copy

- static data member

  Acts as a global variable

- Static data member is declared using the static keyword.
- There is only one copy of the static data member in the class. All the objects share the static data member.
- The static data member is always initialized to zero when the first class object is created.

Syntax:

static data_type datamember_name;

Here

static is the keyword.

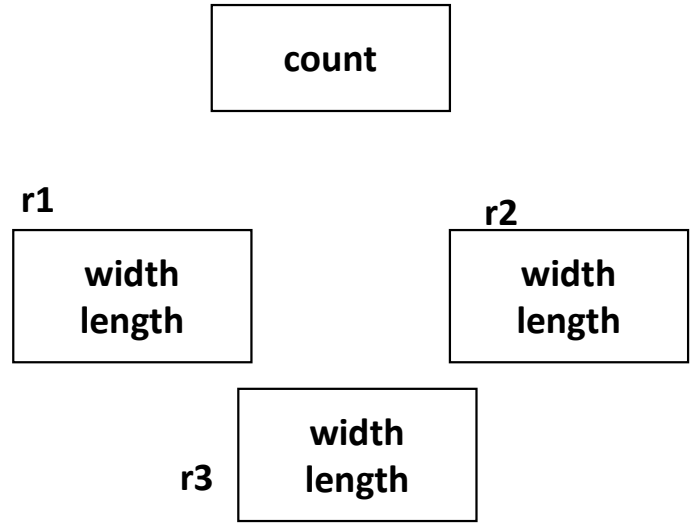
data_type – int , float etc…

datamember_name – user defined

Static Data Member

```
class Rectangle
{
    private:
       int width;
       int length;
       static int count;
    public:
       void set(int w, int
    l);
       int area();
    }
```

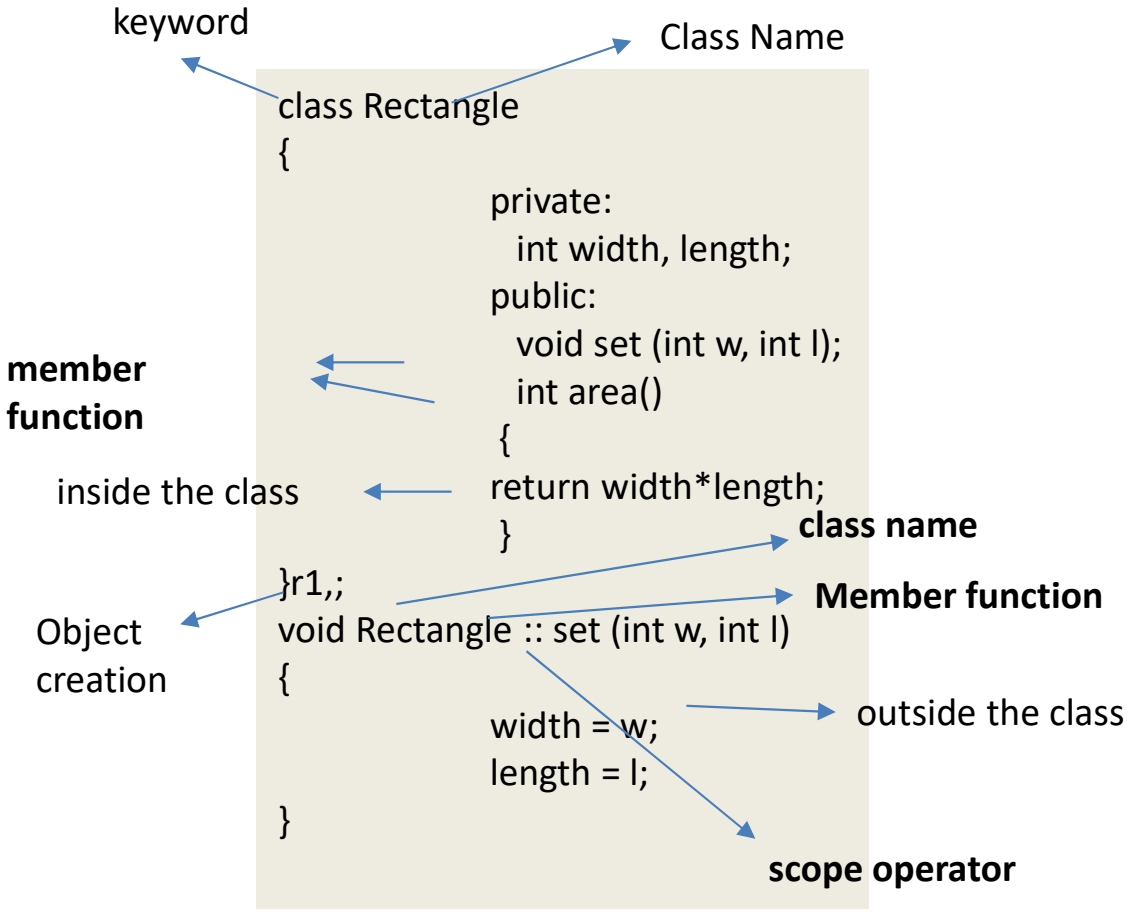
```
Rectangle
    r1;
Rectangle
    r2;
Rectangle
    r3;
```



Member Functions

- Used to
  - access the values of the data members (accessor)
  - perform operations on the data members (implementor)
- Are declared inside the class body
- Their definition can be placed inside the class body, or outside the class body
- Can access both public and private members of the class
- Can be referred to using dot or arrow member access operator

Member function




```
int main()
{
Rectangle r2;
   r1.set(5,8);
   int x=r1.area();
   cout<<"x value in r1 "<<x;
   r2.set(5,8);
   int x=r1.area();
   cout<<"x value in r2 "<<x;

}
```

Object creation Inside the function

const member function

- The const member functions are the functions which are declared as constant in the program.
- The object called by these functions cannot be modified.
- It is recommended to use const keyword so that accidental changes to object are avoided.
- A const member function can be called by any type of object.
- *Note: Non-const functions can be called by non-const objects only.*

Syntax:

User defined

keyword

```
datatype function_name const();
```

Const Member Function

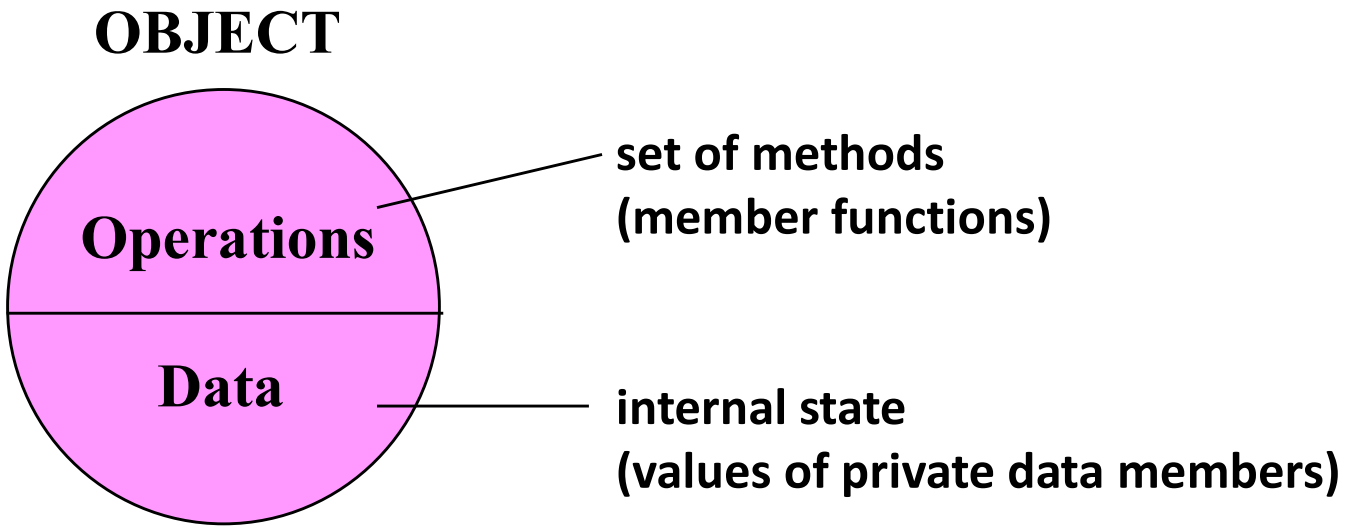
```
class  Time
{
 private :
    int     hrs, mins, secs ;

 public :
    void       Write ( )  const ;
} ;
```

function declaration

function definition

```
void Time :: Write( ) const
{
    cout <<hrs << ":" << mins << ":" << secs << endl;
}
```

What is an object?



Object:

- Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.
- Each object has different data variables.
- Objects are initialized using special class functions called **Constructors**.
- **Destructor** is special class member function, to release the memory reserved by the object.

Syntax of creating object :

Class_name object_name;

Declaration of an Object

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       void set(int w, int l);
       int area();
};
```

```
main()
{
     Rectangle r1;
     Rectangle r2;

    r1.set(5, 8);
     cout<<r1.area()<<endl;

     r2.set(8,10);
     cout<<r2.area()<<endl;
}
```

Another Example

```
#include <iostream.h>

class circle
{
      private:
         double radius;

      public:
         void store(double);
         double area(void);
         void display(void);

};
```

```
// member function definitions

void circle::store(double r)
{
   radius = r;
}
double circle::area(void)
{
   return 3.14*radius*radius;
}
void circle::display(void)
{
   cout << "r = " << radius <<
     endl;
}
```

```
int main(void) {
   circle c;   // an object of circle class
   c.store(5.0);
   cout << "The area of circle c is " << c.area() << endl;
   c.display();
}
```

Declaration of an Object

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       void set(int w, int l);
       int area();
};
```

r1 is statically allocated

```
main()
{
      Rectangle r1;
⇨ r1.set(5, 8);
}
```

r1

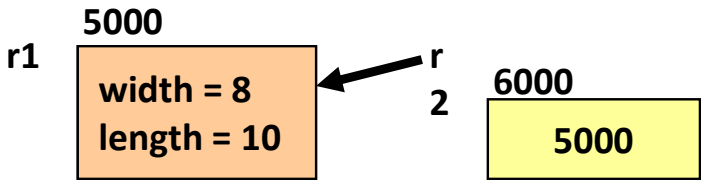
width = 5
length = 8

Declaration of an Object

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

r2 is a pointer to a Rectangle object

```
main()
{
    Rectangle r1;
    r1.set(5, 8);        //dot notation

    Rectangle *r2;
    r2 = &r1;
    r2->set(8,10);       //arrow notation
}
```



Object Initialization

1. By Assignment

```
#include <iostream.h>

class circle
{
    public:
        double radius;
};
```

- Only work for public data members
- No control over the operations on data members

```
int main()
{
   circle c1;              // Declare an instance of the class circle
   c1.radius = 5;          // Initialize by assignment

}
```

Object Initialization

2. By Public Member Functions

```
#include <iostream.h>

class circle
{
  private:
      double radius;

   public:
      void set (double r)
                {radius = r;}
      double get_r ()
                {return radius;}

};
```

```
int main(void) {
    circle c;                    // an object of circle class
    c.set(5.0);                  // initialize an object with a public member
      function
    cout << "The radius of circle c is " << c.get_r() << endl;
        // access a private data member with an accessor
}
```

Declaration & Initialization of an Object

```
#include <iostream.h>

class circle
{
    public:
        double radius;
};
```

- Only work for public data members
- No control over the operations on data members

```
int main()
{
   circle c1;              // Declare an instance of the class circle
   c1.radius = 5;          // Initialize by assignment

}
```

Examples

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       void set(int w, int l);
       int area();
};
```

```
main()
{
     Rectangle r1;
     Rectangle r2;

    r1.set(5, 8);
     cout<<r1.area()<<endl;

     r2.set(8,10);
     cout<<r2.area()<<endl;
}
```

Example: 2

```
// Program to illustrate the working of
// objects and class in C++ Programming

#include <iostream>
using namespace std;

// create a class
class Room {

  public:
   double length;
   double breadth;
   double height;

   double calculateArea() {
      return length * breadth;
   }

   double calculateVolume() {
      return length * breadth * height;
   }
};

int main() {

   // create object of Room class
   Room room1;

   // assign values to data members
   room1.length = 42.5;
   room1.breadth = 30.8;
   room1.height = 19.2;

   // calculate and display the area and
volume of the room
   cout << "Area of Room =  " <<
room1.calculateArea() << endl;
   cout << "Volume of Room =  " <<
room1.calculateVolume() << endl;

   return 0;
}
```

Session-4

Feature : Objects and Classes

Objects and Classes

- Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance of that class.

–**Attributes** – member of a class.

- An attribute is the data defined in a class that maintains the current state of an object. The state of an object is determined by the current contents of all the attributes.

–**Methods** – member functions of a class

Objects and classes

- A class itself does not exist; it is merely a description of an object.
- A class can be considered a template for the creation of object

- Blueprint of a building → class
- Building → object

- An object exists and is definable.
- An object exhibits behavior, maintains state, and has traits. An object can be manipulated.
- An object is an instance of a class.

Methods

- A ***method*** is a function that is part of the class definition. The methods of a class specify how its objects will respond to any particular message.
- A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything.
- Methods allow us to reuse the code without retyping the code.
- A *message* is a request, sent to an object, that activates a method (i.e., a member function).

Defining a Base Class - Example

•A base class is not defined on, nor does it inherit members from, any other class.

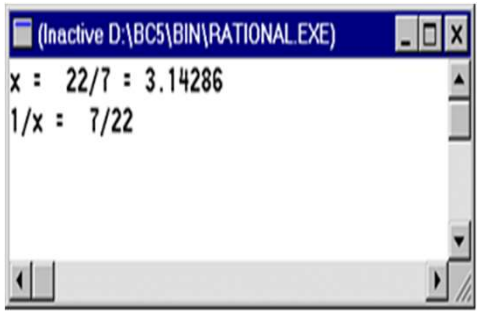
```
#include <iostream>
using std::cout;      //this example "uses" only the necessary
using std::endl;      // objects, not the entire std namespace


class Fraction {
public:
    void assign (int, int);              //member functions
    double convert();
    void invert();
    void print();
private:
     int num, den;                       //member data
};
```

Continued…

Using objects of a Class - Example

The main() function:

```
int main()
{
   Fraction x;
   x.assign (22, 7);
   cout << "x =  "; x.print();
   cout << " = " << x.convert() << endl;
   x.invert();
   cout << "1/x =  "; x.print(); cout << endl;
   return 0;
}
```



Continued…

Defining a Class – Example

Class Implementation:

```
void Fraction::assign (int numerator, int denominator)
{
    num = numerator;
    den = denominator;
}

double Fraction::convert ()
{
    return double (num)/(den);
}

void Fraction::invert()
{
    int temp = num;
    num = den;
    den = temp;
}

void Fraction::print()
{
    cout << num << '/' <
}< den;
```

Difference Between Class and Structure

S.No	Class	Structure
1	Members of a class are private by default.	Members of a structure are public by default.
2	Member classes/structures of a class are private by default.	Member classes/structures of a structure are public by default.
3	It is normally used for data abstraction and further inheritance.	It is normally used for the grouping of data
4	Security is high here.	Less Security.
5	It is declared using the **class** keyword.	It is declared using the **struct** keyword.

Debug the code

```
/*Beginning of structDistance02.cpp*/
#include <iostream.h>
struct Distance {
private: int iFeet; float fInches;
public: void setFeet(int x) {
iFeet=x; //????? Legal or not
int getFeet()
{ return iFeet; }
void setInches(float y) {
 fInches=y; }
float getInches() {
 return fInches; } };
```

Debug the code

```
void main() {
Distance d1,d2;
d1.setFeet(2);
d1.setInches(2.2);
 d2.setFeet(3);
d2.setInches(3.3);
 d1.iFeet++; //  ??????
cout<<d1.getFeet()<<d1.getInches()<<d2.getFeet()<<d2.getInche
s()<<endl;
```

Scope Resolution Operator

- It is possible and usually necessary for the library programmer to define the member functions outside their respective classes.
- The scope resolution operator makes this possible.
- Program illustrates the use of the scope resolution operator (::)

Scope Resolution Operator Example

```
/*Beginning of scopeResolution.cpp*/
class Distance
{
 int iFeet;
 float fInches;
public:
 void setFeet(int); //prototype only
 int getFeet(); //prototype only
 void setInches(float); //prototype only
 float getInches(); //prototype only
};
void Distance::setFeet(int x) //definition {  iFeet=x; }
int Distance::getFeet() //definition { return iFeet; }
void Distance::setInches(float y) //definition {  fInches=y; }
float Distance::getInches() //definition{return fInches;}
/*End of scopeResolution.cpp*/
```

A Recap……

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


class Fraction {
public:
                void assign
(int, int);
                double
convert();
   void invert();
                void print();
private:
                 int num,
den;
};
```

```
void Fraction::assign (int
numerator, int denominator)
{
                num = numerator;
                den = denominator;
}

double Fraction::convert ()
{
                return double
(num)/(den);
}

void Fraction::invert()
{
                int temp = num;
                num = den;
                den = temp;
}

void Fraction::print()
{
                cout << num << '/'
<< den;
}
```

```
int main()
{
                Fraction x;
                x.assign (22, 7);
                cout << "x =  "; x.print();
                cout << " = " << x.convert() <<
endl;
                x.invert();
                cout << "1/x =  "; x.print(); cout
<< endl;
                return 0;
}
```

Session 3 ,7

Topic : UML Class Diagram, its components, Class Diagram relations and Multiplicity

UML Diagrams - Introduction

- UML stands for **U**nified **M**odeling **L**anguage
- It is a modern approach to ***modeling and documenting*** software
- It is one of the most popular ***business process modeling*** techniques
- It is based on ***diagrammatic representations*** of software components
- An old proverb says, *"A picture is worth a thousand words"*
- By using visual representations, we are able to ***better understand possible flaws or errors*** in software or business processes

UML Diagrams - Introduction

SKETCH

- In this case, UML diagrams are only a ***lop-level view*** of the system
- It may not include all the necessary details to execute the project
- **Forward Design:**
  - The design of the sketch is done before coding the application
  - We can get a better view of the system or workflow that we are trying to create
  - Many design issues or flaws can be revealed
  - The overall project health and well-being can be improved
- **Backward Design:**
  - After writing the code, the UML diagrams are drawn as a form of documentation

UML Diagrams - Introduction

BLUEPRINT

- In this case, the UML diagram serves as a complete design that requires the actual implementation of the system or software.
- Often, this is done by using **CASE tools** (**C**omputer **A**ided **S**oftware **E**ngineering Tools)
- It needs more expertise and user training

Overview of UML Diagrams

Structural

: element of spec. irrespective of time

- Class
- Component
- Deployment
- Object
- *Composite structure*
- *Package*

Behavioral

: behavioral features of a system / business process

- Activity
- State machine
- Use case
- *Interaction*

Interaction

: emphasize object interaction

- Communication(collaberation)
- Sequence
- *Interaction overview*
- *Timing*

Class Diagram

- It is the most common diagram type for software documentation
- Class diagrams contain classes with their attributes (data members) and their behaviours (member functions)
- Each class has 3 fields:
  - Class Name
  - Attributes
  - Behaviours
- The relation between different classes makes up a class diagram

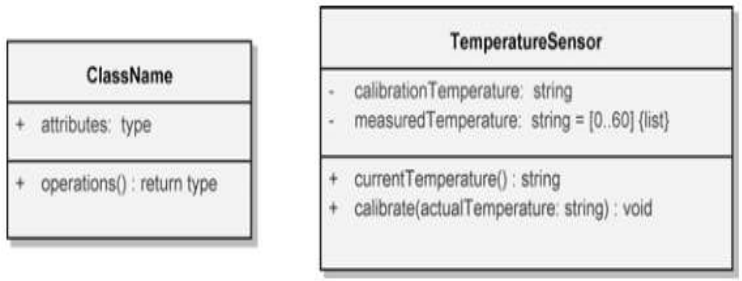
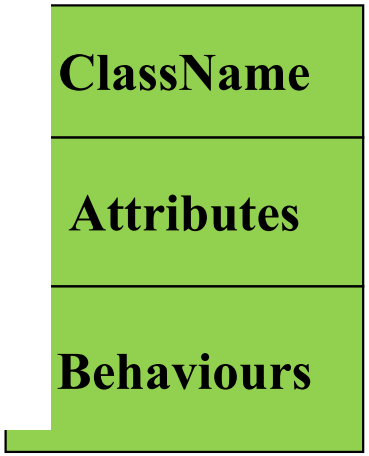
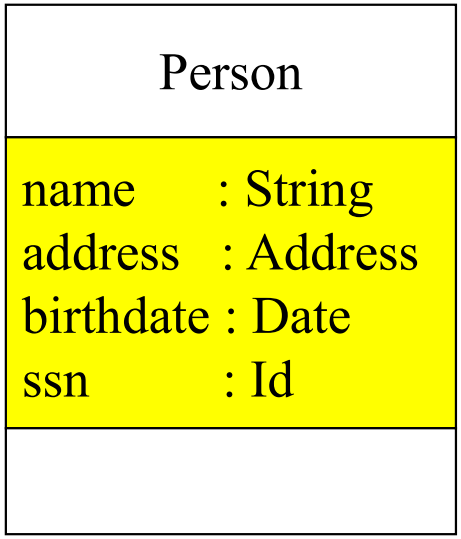


Figure 5–33 A General Class Icon and an Example for the Gardening System

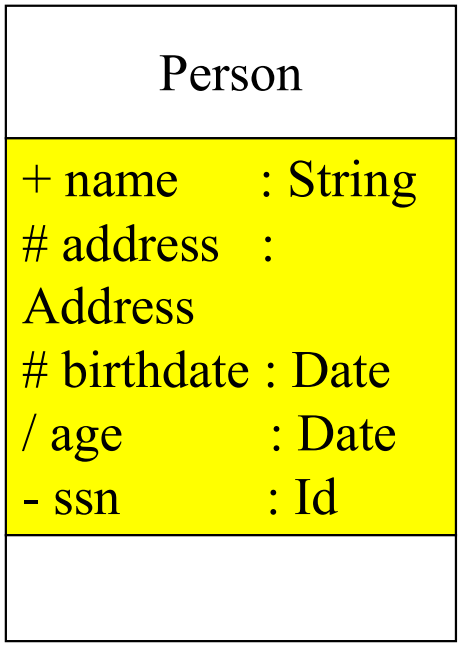


Class Attributes



An *attribute* is a named property of a class that describes the object being modeled. In the class diagram, attributes appear in the second compartment just below the name-compartment.

Class Attributes (Cont'd)



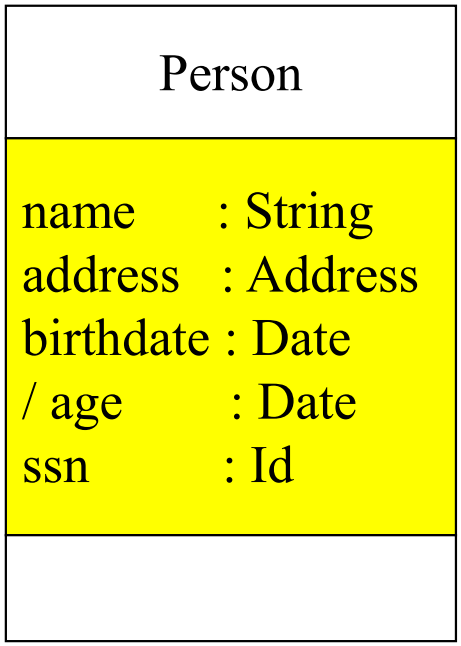
Attributes are usually listed in the form:

attributeName : Type

Attributes can be:

+ public
# protected
- private
/ derived

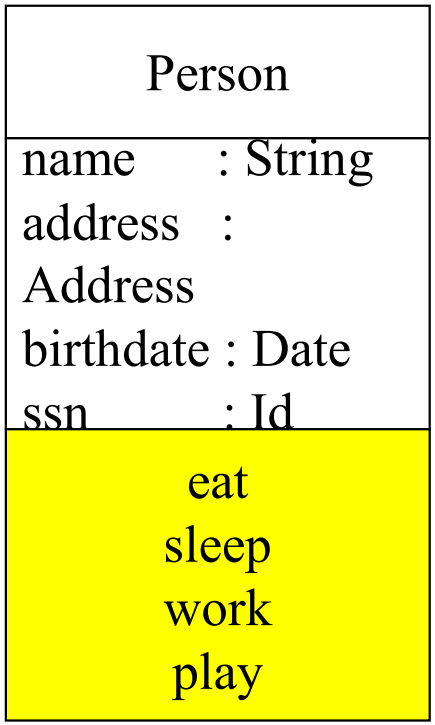
Class Attributes (Cont'd)



A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age can be computed from his birth date. A derived attribute is designated by a preceding '/' as in:

/ age : Date

Class Operations



Operations describe the class behavior and appear in the third compartment.

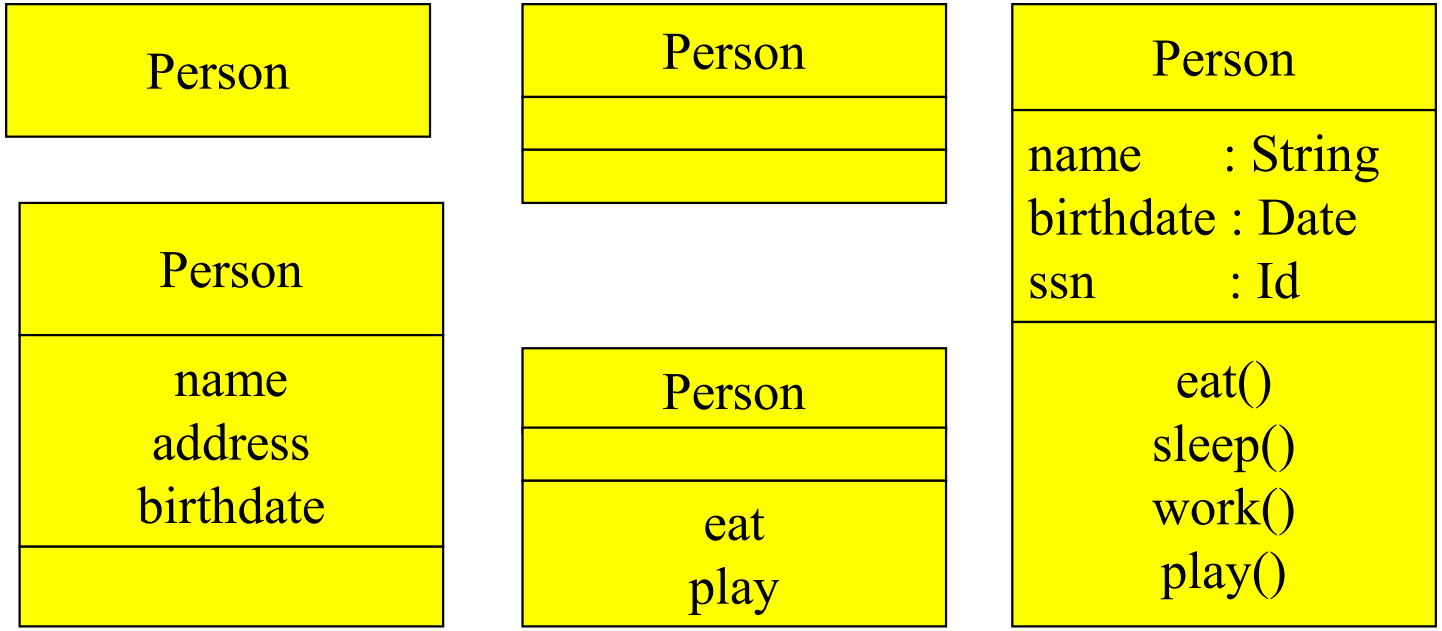
Class Operations (Cont'd)

PhoneBook
newEntry (n : Name, a : Address, p : PhoneNumber, d : Description) getPhone (n : Name, a : Address) : PhoneNumber

You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.

Depicting Classes

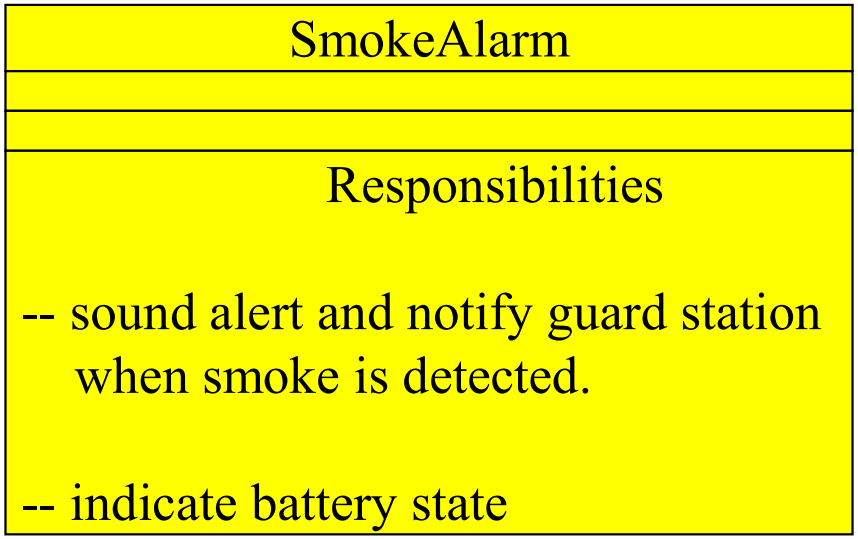
When drawing a class, you needn't show attributes and operation in every diagram.



Class Responsibilities

A class may also include its responsibilities in a class diagram.

A responsibility is a contract or obligation of a class to perform a particular service.



Basic components of a class diagram

The standard class diagram is composed of three sections:

Upper section: Contains the name of the class. This section is always required, to know whether it represents the classifier or an object.

Middle section: Contains the attributes of the class. Use this section to describe the qualities of the class. This is only required when describing a specific instance of a class.

Bottom section: Includes class operations (methods). Displayed in list format, each operation takes up its own line. The operations describe how a class interacts with data.

RULES TO BE FOLLOWED:

- Class name must be unique to its enclosing namespace.
- The class name begins in uppercase and the space between multiple words is omitted.
- The first letter of the attribute and operation names is lowercase with subsequent words starting in uppercase and spaces are omitted.
- Since the class is the namespace for its attributes and operations an attribute name must be unambiguous in the context of the class.

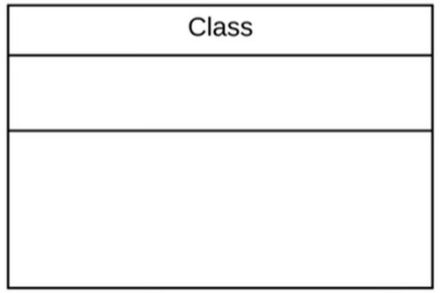
Class Diagram

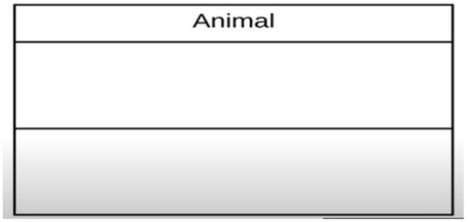
- **Basic Components of a Class Diagram**
  - ***Upper Section***

    It contains the name of the class
  - ***Middle Section***

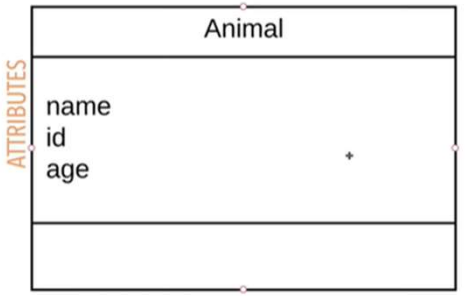
    It contains the attributes of the class
  - ***Bottom Section***

    It includes methods
- Let us model a ***Zoo System***
- What's in a Zoo? Animals
- Let us name the class Animal
- If our class is Animal, an instance of that class would be a specific animal





Class Diagram – cont..



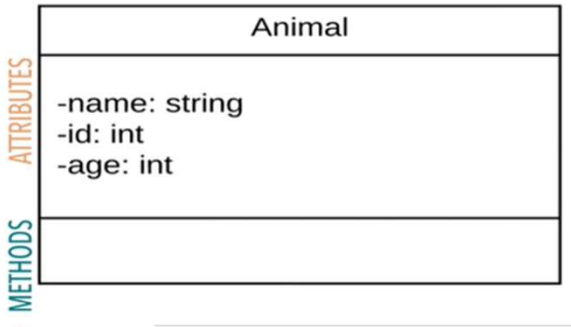
Attributes : Second Section

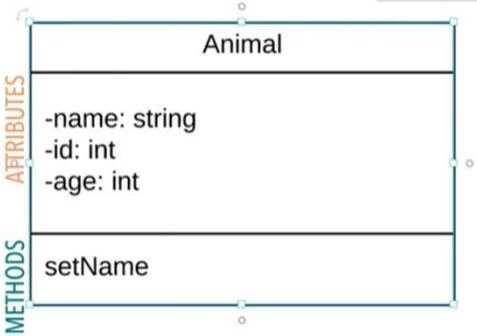
- How would you identify each instance of that class, i.e., a specific animal?
  - Using Attributes
- What are ***Attributes***?
  - A significant piece of data containing values that describe each instance of that class
  - They are also known as fields, variables or properties
- Format for specifying an attribute:

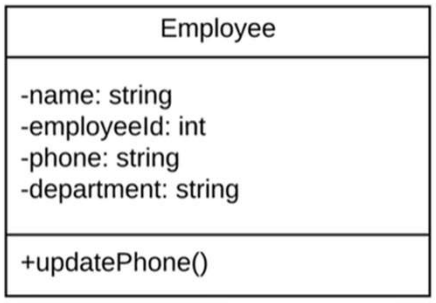
  visibility attributeName : Type [multiplicity] = DefaultValue {property string}
- **Visibility:** It sets the accessibility of an attribute / method
  - **Private (-)** Visible only to the elements within the same class
  - **Public (+)** Visible to any element that can see the class
  - **Protected (#)** Visible to the elements in the class and its sub class
  - **Package (~)** Visible to elements within the same package

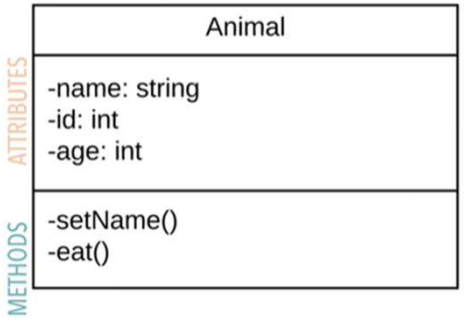
Class Diagram – cont..

- **<u>Methods : Third section</u>**
- They are also known as operations or functions
- It specifies the behaviour of a class
- Eg: We may want to change the name of an Animal
- We could add a method ***setName***
- We could also create a method for eating, since all of our animals eat
- Lets take another example : Employee Details



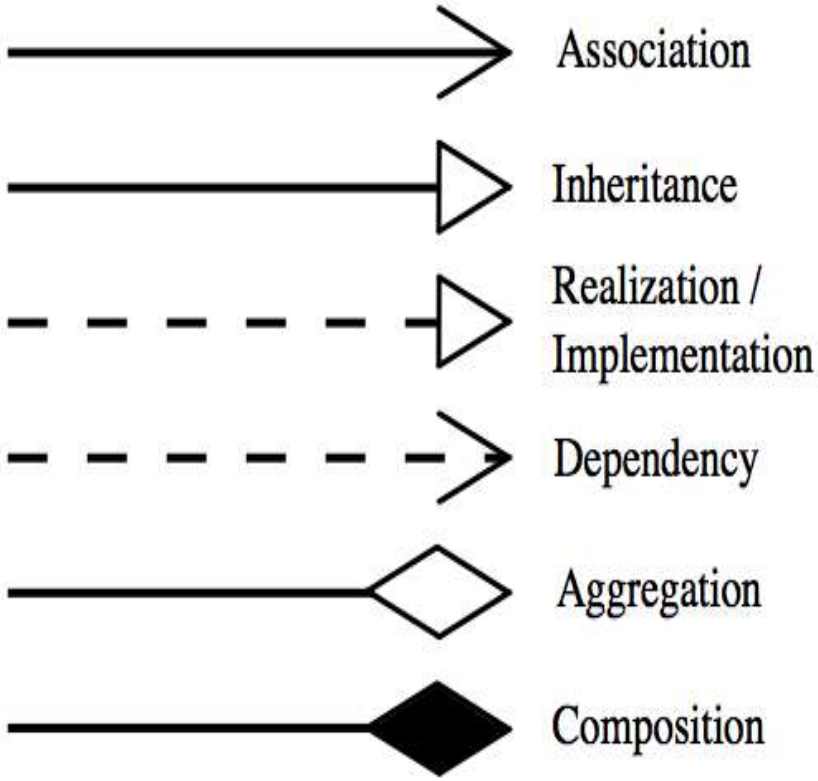






Multiplicity and Relationships

Relationships

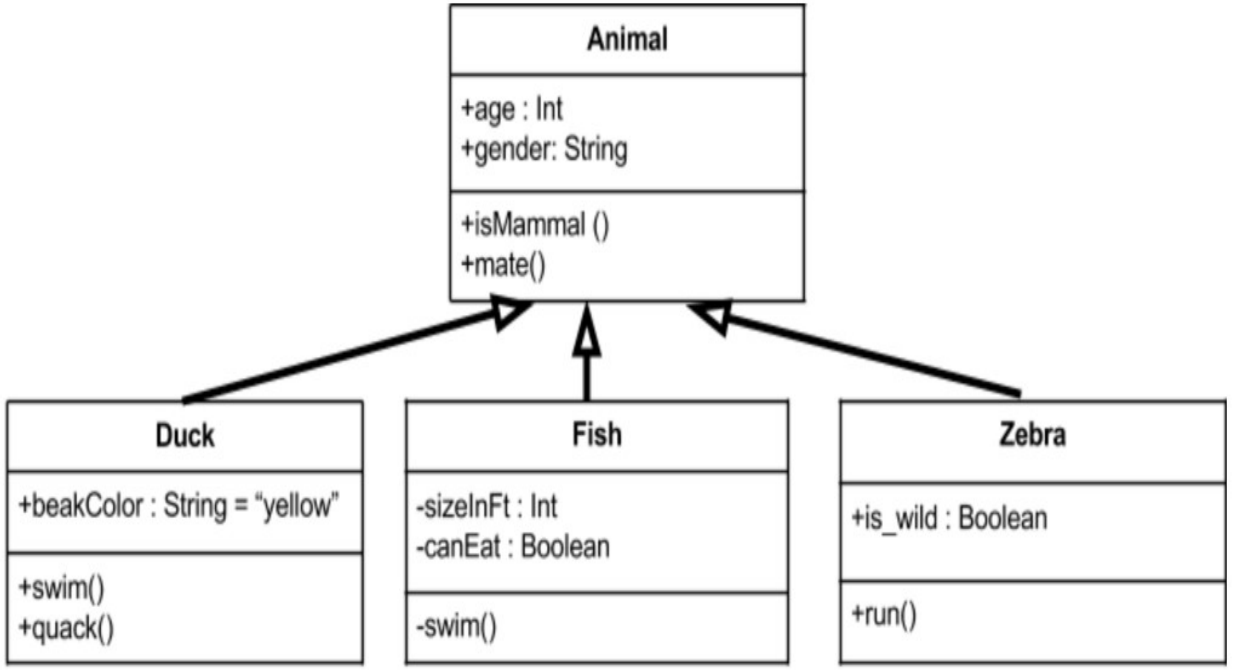


Association

- 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:

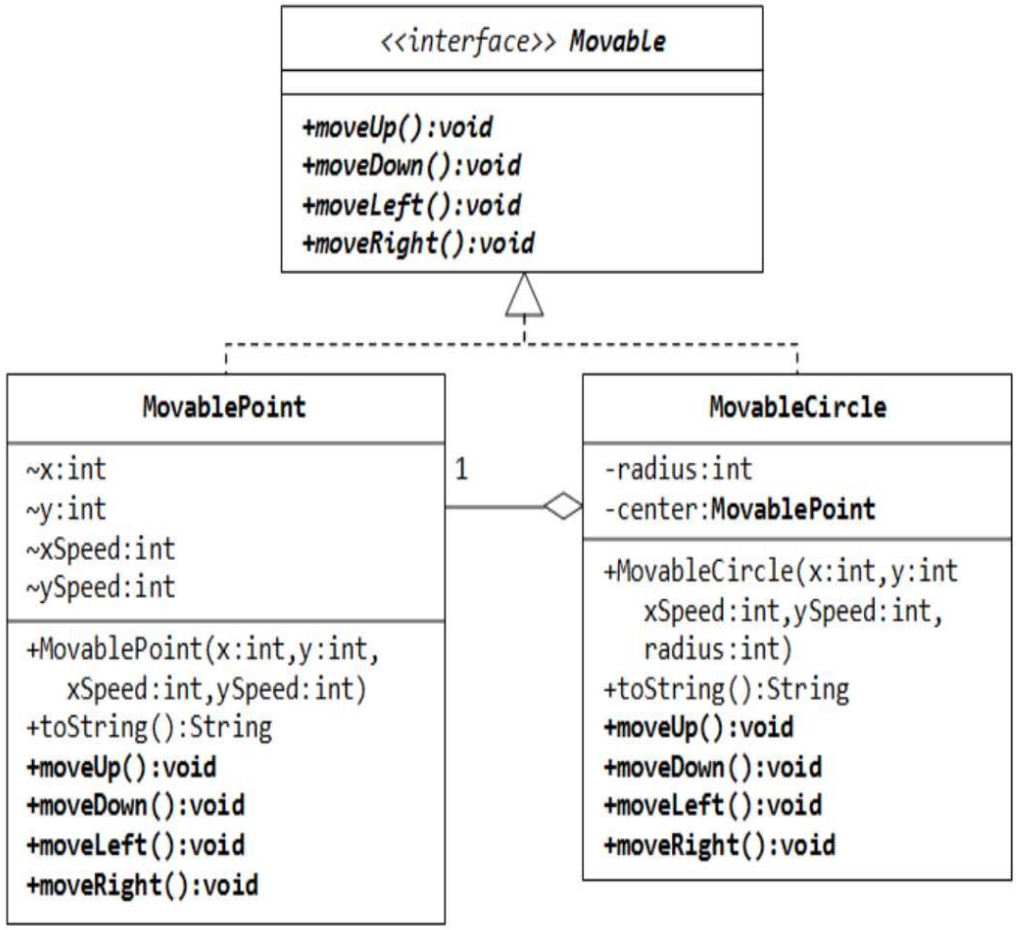


Example

- 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

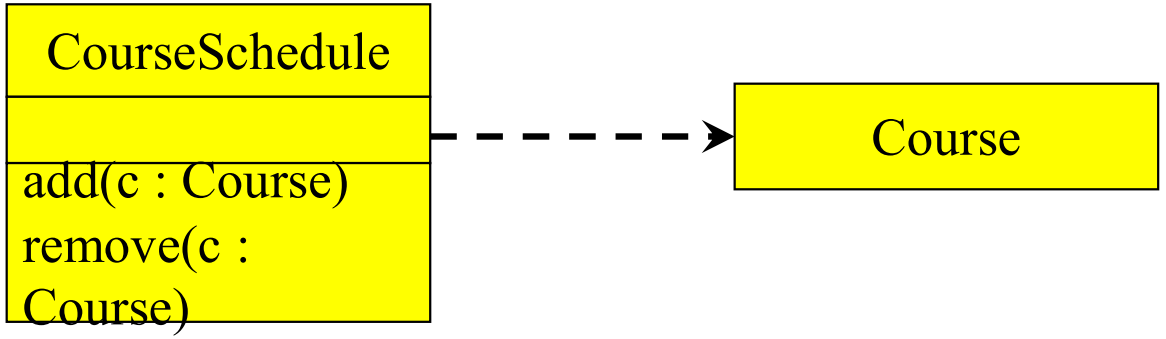
- A relationship between two model elements, in which one model element implements/executes the behavior specifies.



Relationships b/w Classes

Dependency Relationships

A *dependency* indicates a semantic relationship between two or more elements. The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.

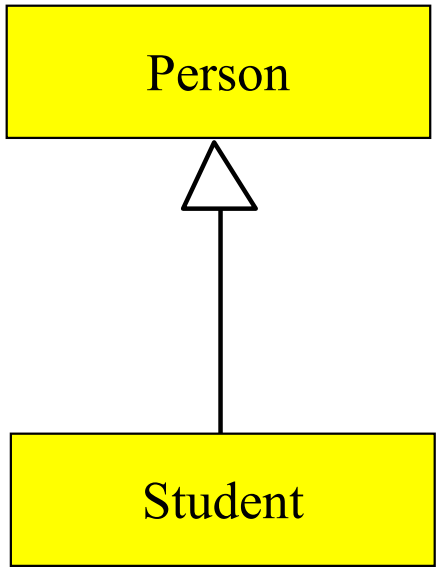


Dependency - Aggregation

- 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.

Relationships b/w Classes

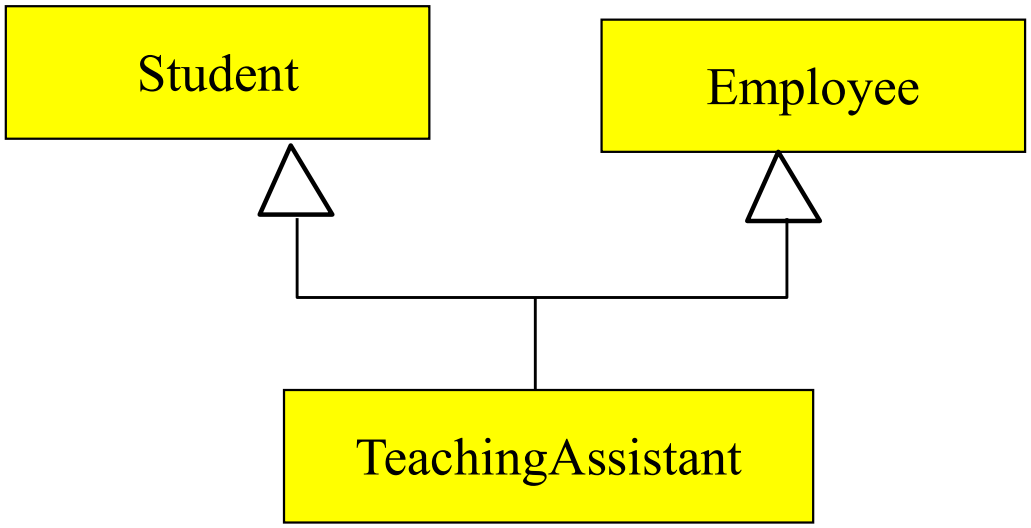
Generalization Relationships



A *generalization* connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

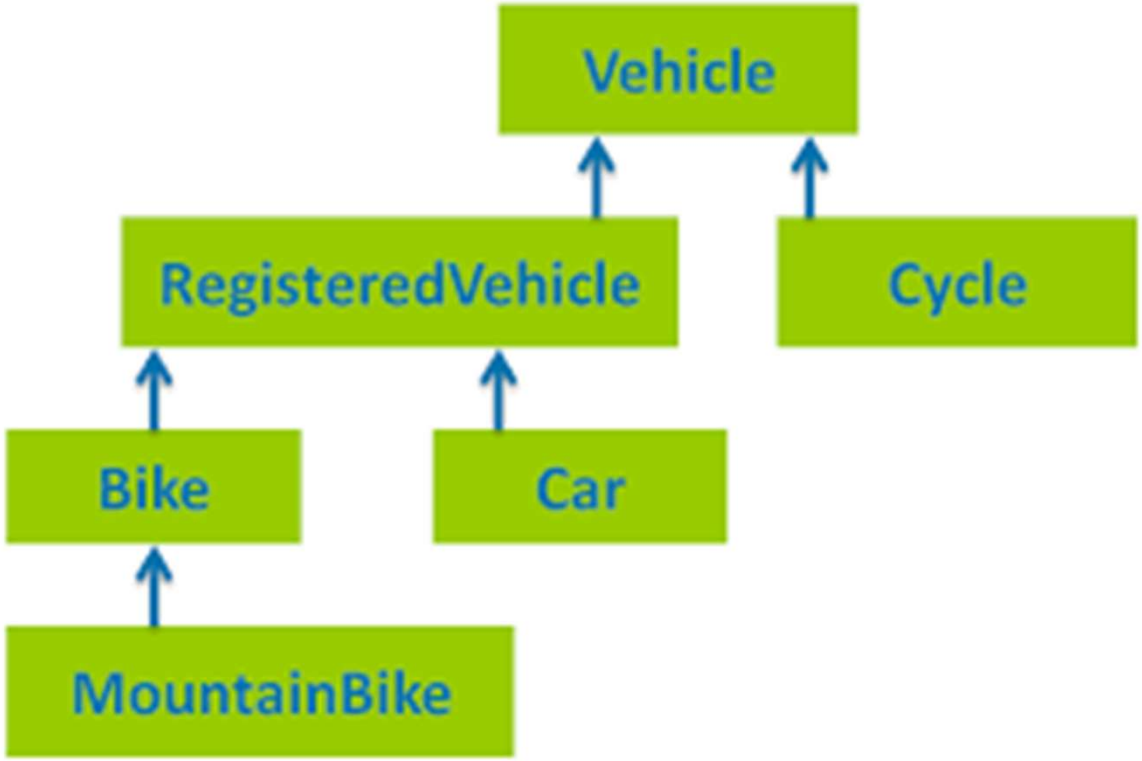
Generalization Relationships (Cont'd)

UML permits a class to inherit from multiple superclasses, although some programming languages (*e.g.,* Java) do not permit **multiple inheritance.**



Generalization Relationships (Cont'd)

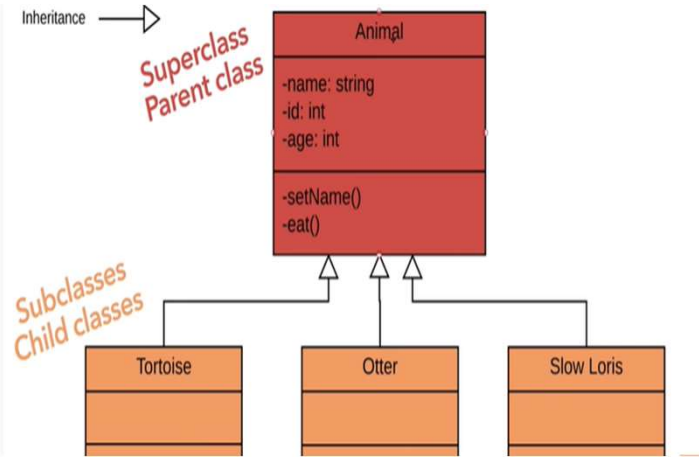
Multilevel Inheritance



Relationships b/w Classes

Inheritance (Generalization)

- Lets say in our zoo, the only animals we have are ***tortoises, otters and Slow Loris***
- We make 3 new classes: Tortoise, Otter and Slow Loris
- These subclasses inherit all the attributes and methods of the superclass
- We could add an attribute specific to Otter in the Otter class
- The main advantage of inheritance is that if we wanted to change or add an attribute for all animals, we need not make change in every sub class

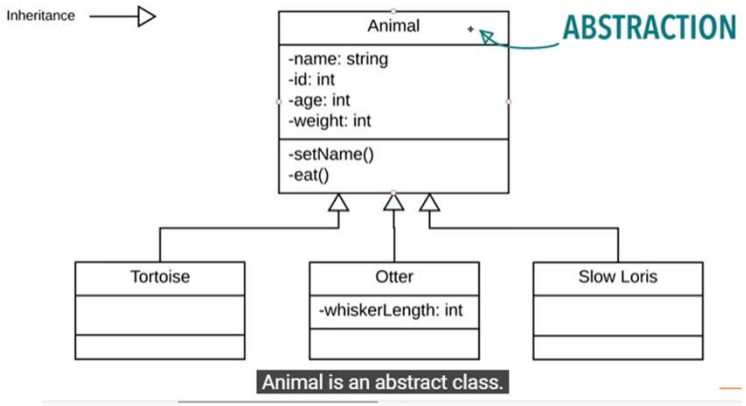


- We just make the change to the base class 'Animal' class and it applies across all sub classes

Relationships b/w Classes

Abstraction

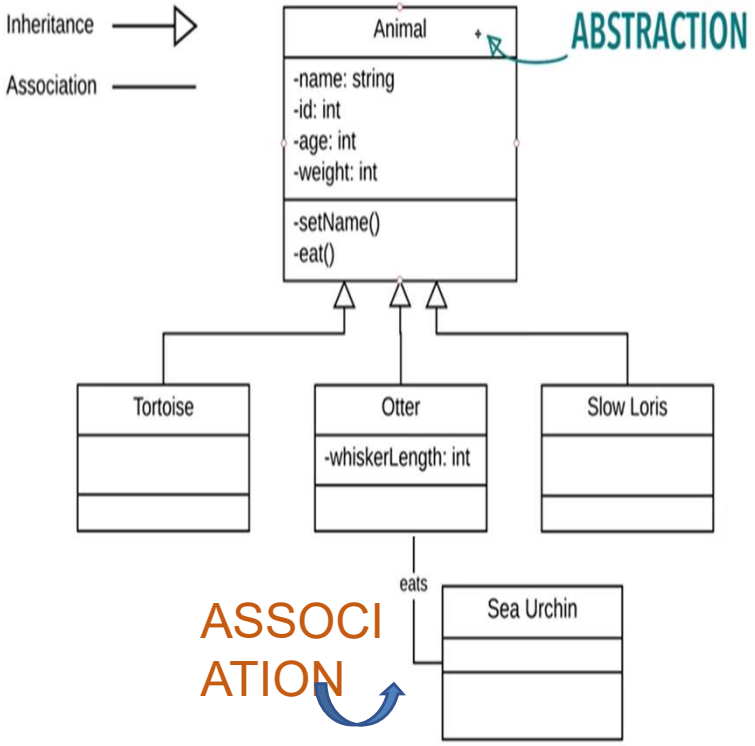
- In this example 'Animal' is the Abstract Class
- We will create instance for only the subclasses
- We wouldn't instantiate the Animal class itself
- The Animal class is just a way to simplify things and keep the code "dry"
- To show that it is an Abstract class, we can write the name of the class in italics ***Animal*** or we can enclose it like ***<<Animal>>***

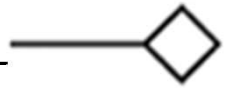


Relationships b/w Classes

Association

- If we had a class for Sea Urchin
  we could draw an association
- Sea otter is a marine mammal in
  the north Pacific Ocean
- They eat mostly hard-shelled invertebrates
  including sea urchins.

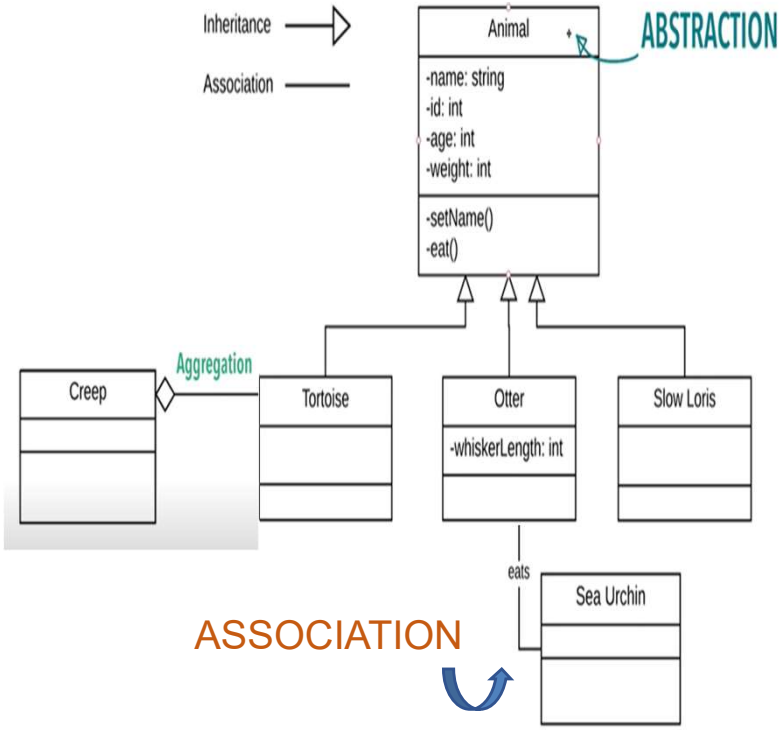


Aggregation  **(open diamond)**

- It is a special type of association that specifies a whole and its parts, where a part can exist outside the whose
- Lets take a group of tortoises, called creep

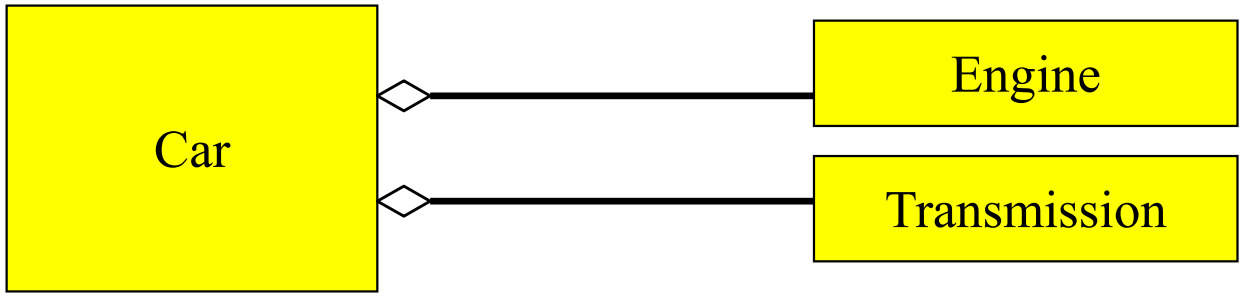


- Any of our zoo's tortoises could be part of a creep



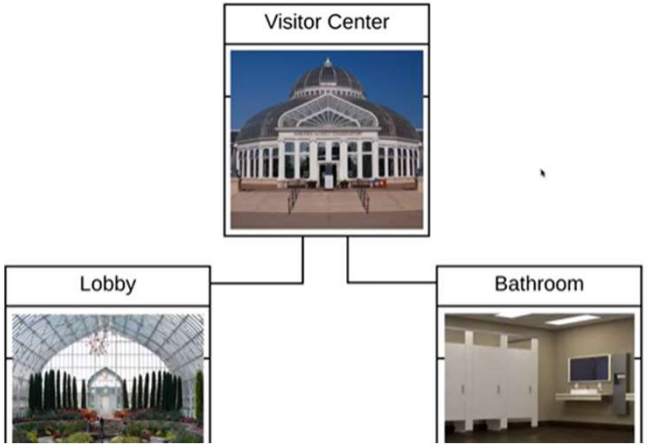
A tortoise could leave the creep at any point and still Exist on its own

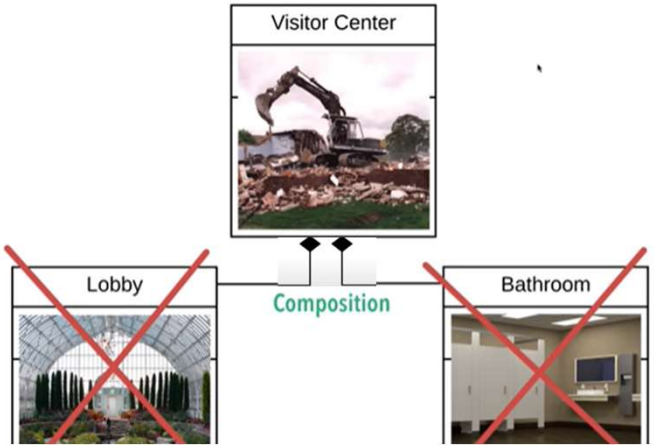

An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.



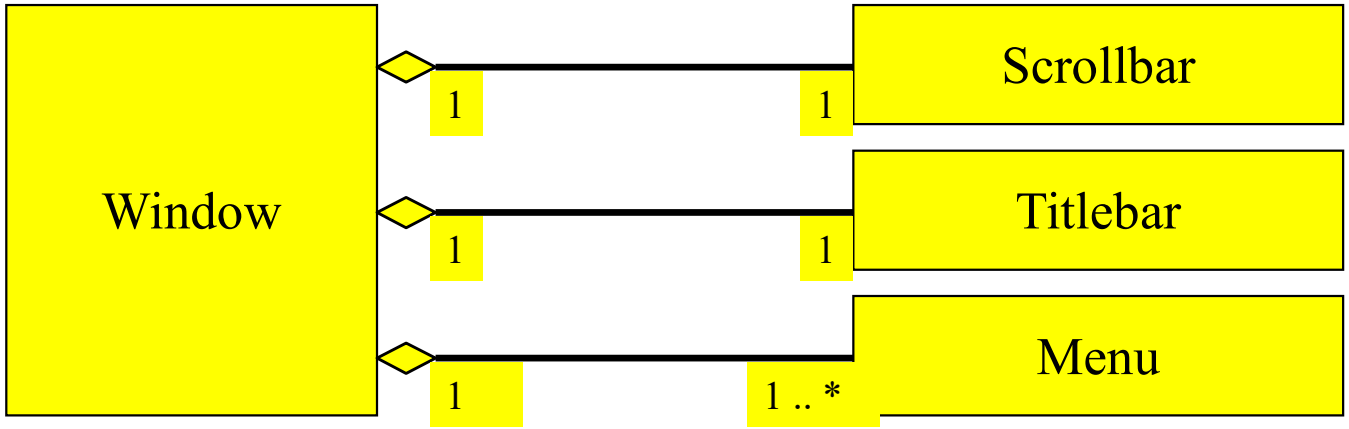
Composition

- It is a special type of association that specifies a whole and its parts, where the part cannot exist outside the whole
- Example: Let us say we have several different visitors centers in our zoo. And each of those visitor centers has a lobby and a bathroom
- Now if one of our visitors centers was torn down, the lobby and the bathroom of that visitor center are torn down as well.
- **Composition :** A child object cannot exist without its parent object



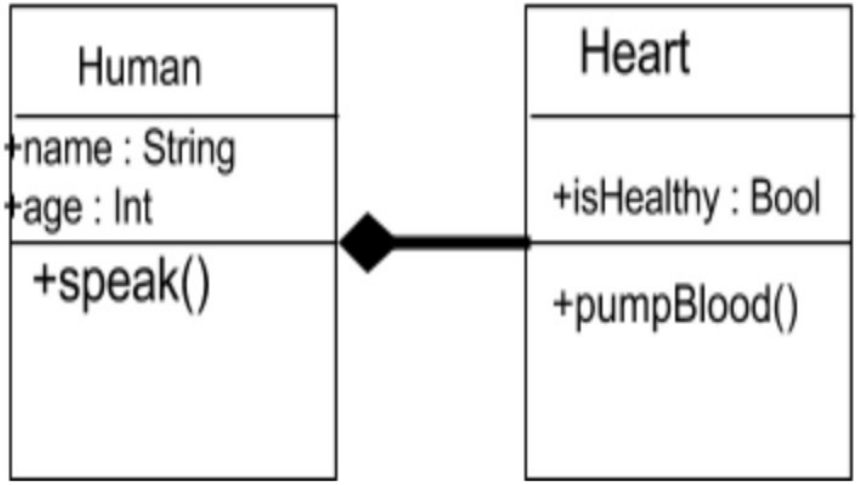


A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.,* they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



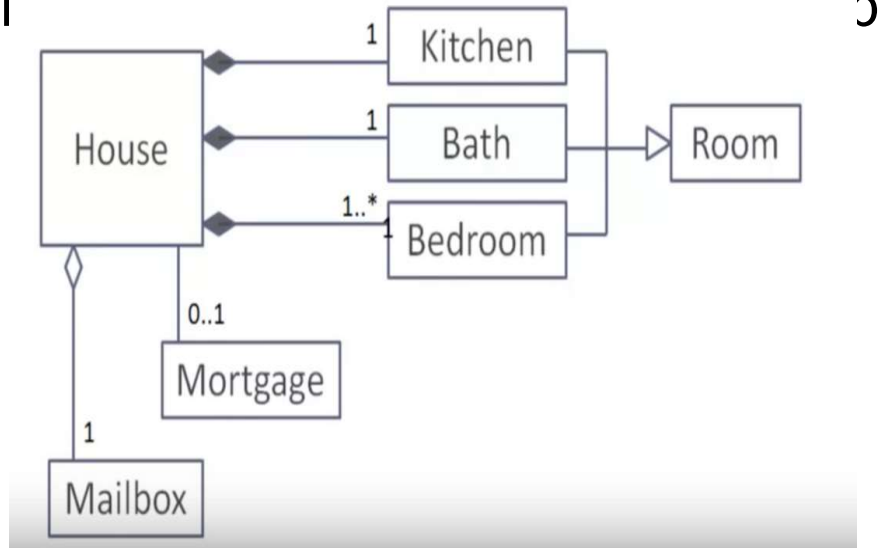
Composition

- 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 dies then ano mposition.

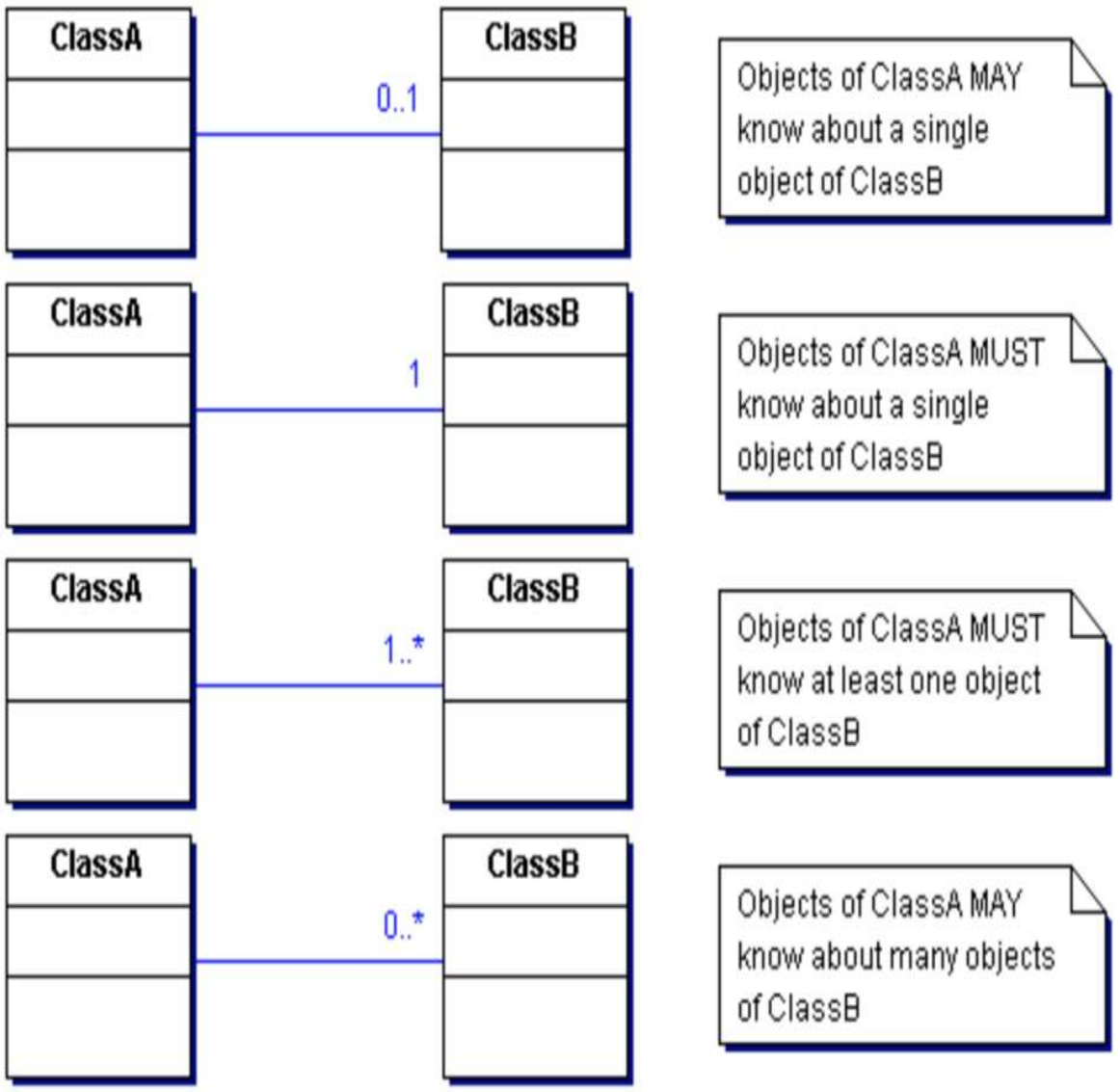


Multiplicity

- After specifying the type of association relationship by connecting the classes, you can also declare the cardinality between tl ɔle:



- 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).

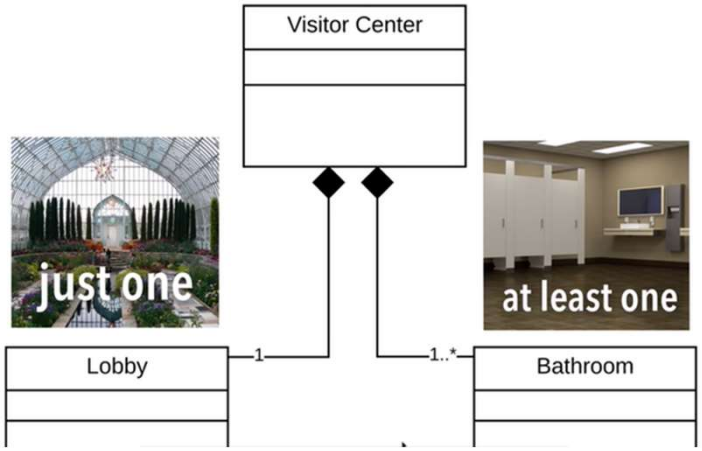

ClassA
ClassB
0..1
Objects of ClassA MAY know about a single object of ClassB
ClassA
ClassB
1
Objects of ClassA MUST know about a single object of ClassB
ClassA
ClassB
1..*
Objects of ClassA MUST know at least one object of ClassB
ClassA
ClassB
0..*
Objects of ClassA MAY know about many objects of ClassB

Multiplicity Constraints

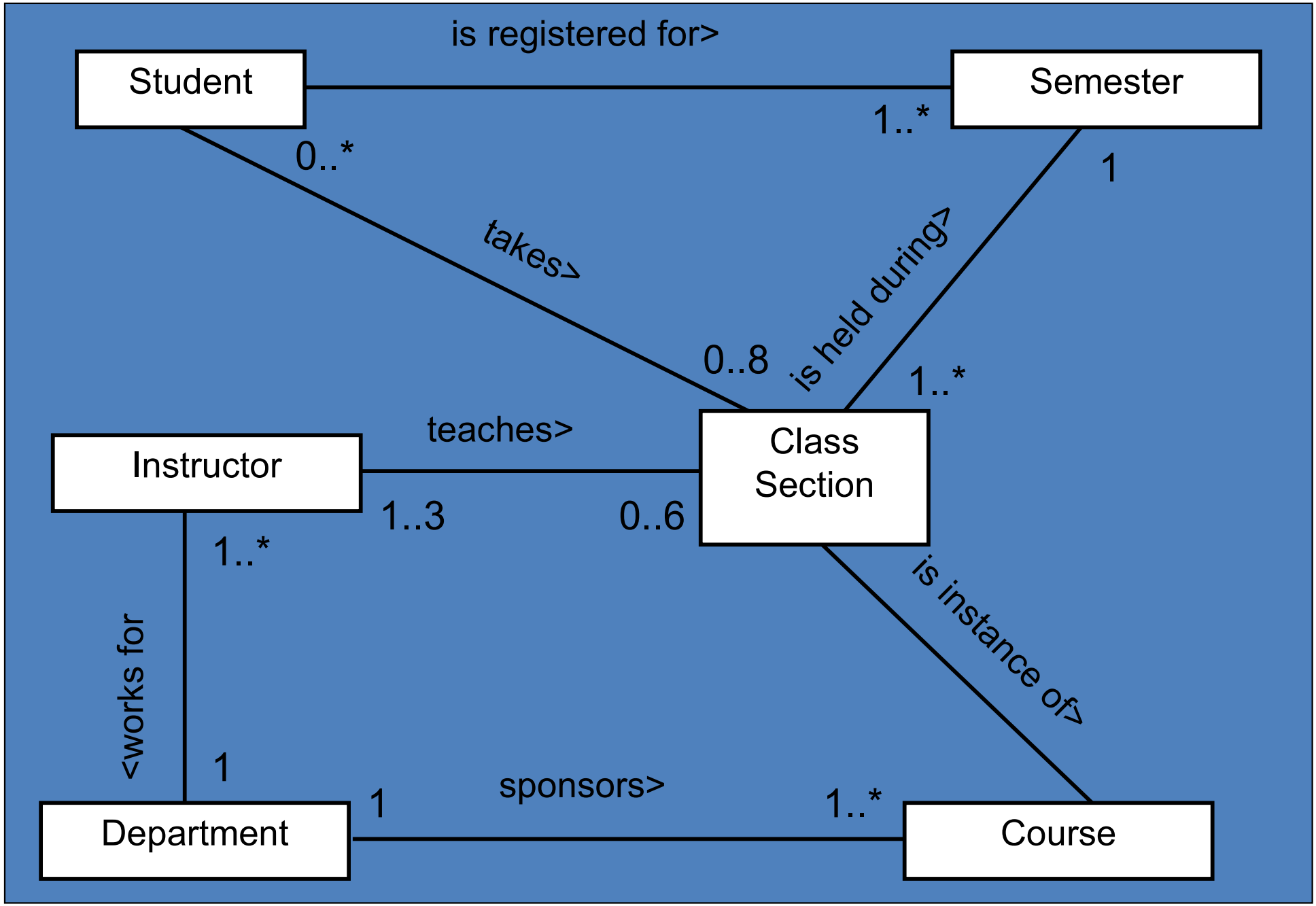
- A multiplicity constraint can be
  - a single number
  - a “*”, meaning an arbitrarily large number or simply “many”
  - a range, denoted by “min..max”
  - a sequence of single numbers and ranges

Multiplicity

- It allows us to set numerical constraints on relationships
- Some types of multiplicity are
- 1 – Just one
- n – Specific number
- 0..* - Zero to many
- 1..* - One to many (At least one)
- 0..1 – Zero or one (Optional)
- m..n – Specific number range



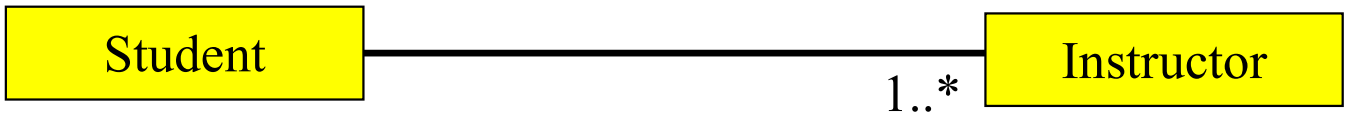
Multiplicity Constraints



Association Relationships (Cont'd)

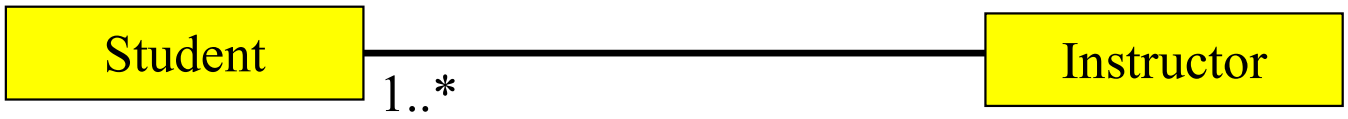
We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association.

The example indicates that a *Student* has one or more *Instructors*:



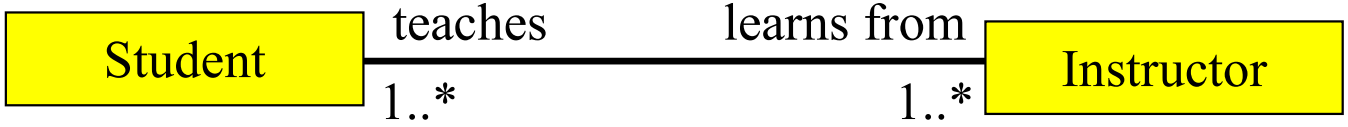
Association Relationships (Cont'd)

The example indicates that every *Instructor* has one or more *Students*:



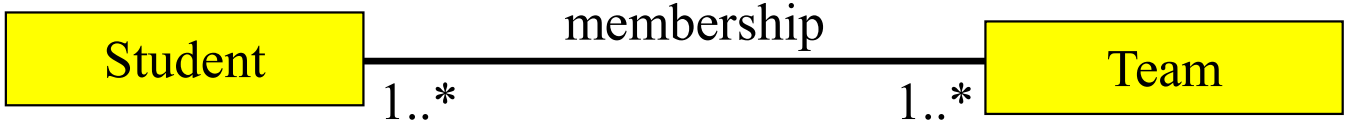
Association Relationships (Cont'd)

We can also indicate the behavior of an object in an association (*i.e.,* the *role* of an object) using *rolenames.*



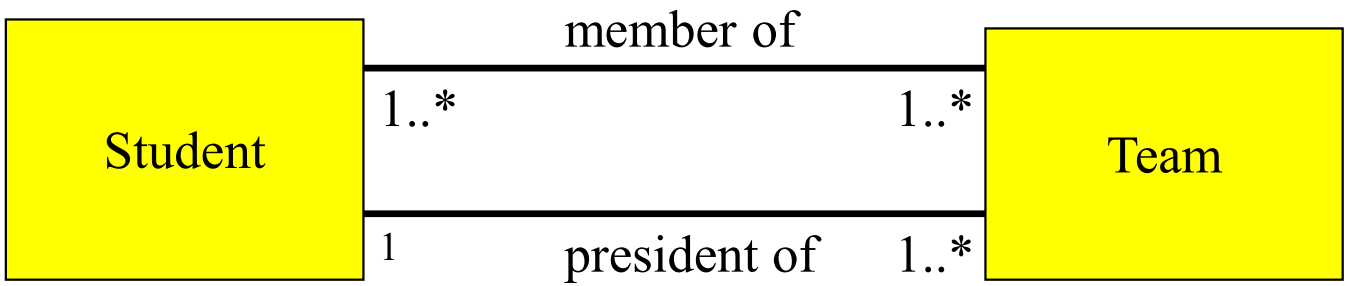
Association Relationships (Cont'd)

We can also name the association.



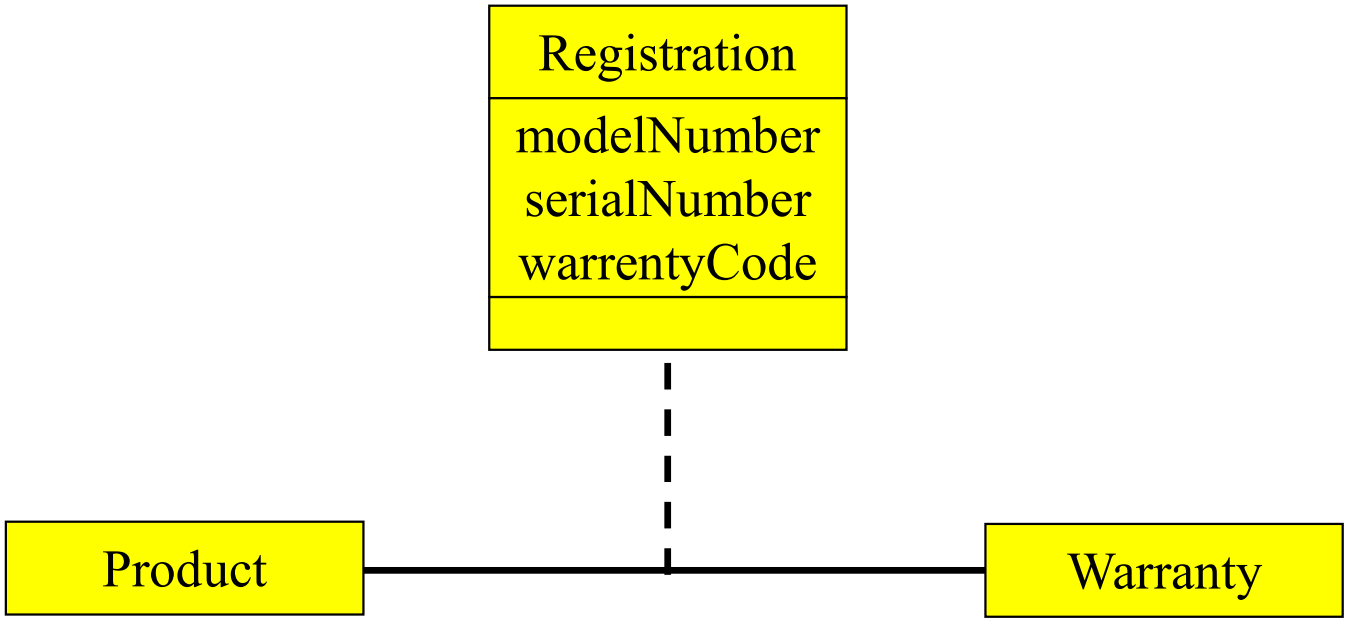
Association Relationships (Cont'd)

We can specify dual associations.



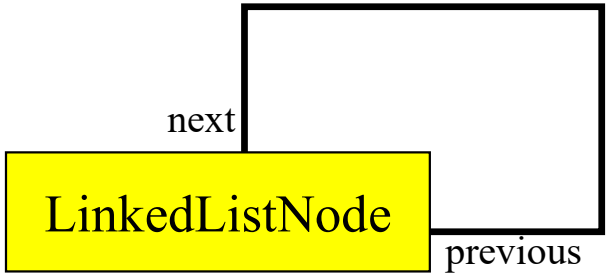
Association Relationships (Cont'd)

Associations can also be objects themselves, called *link classes* or an *association classes*.



Association Relationships (Cont'd)

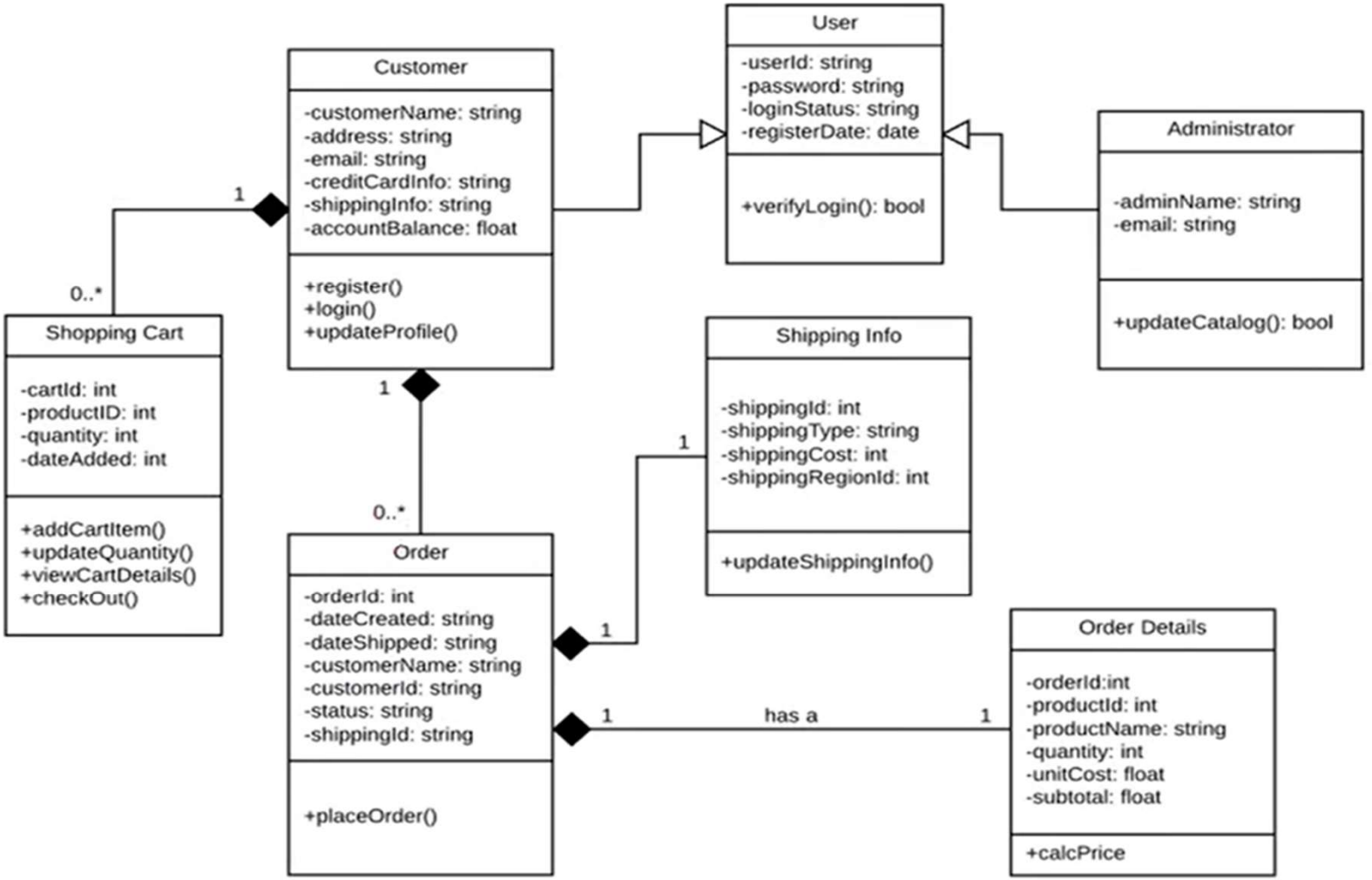
A class can have a *self association*.



Interfaces

An *interface* is a named set of operations that specifies the behavior of objects without showing their inner structure. It can be rendered in the model by a one- or two-compartment rectangle, with the *stereotype* <<interface>> above the interface name.

A real world Example



A Simple C++ Program with class

```
#include<iostream>
using namespace std;
class product
{
    private:
        int number;
        float cost;
    public:
        void getdata(int a, float b);
        void putdata(char c)
        {
            cout<<c<<"\t"<<number<<"\t"<<cost<<"\n";
        }
};
void product::getdata(int a, float b)
{
    number=a;
    cost=b;
}
```

```
int main()
{
    int n;
    float c;
    product x;
    cout<<"Enter the details of product X \n";
    cout<<"Enter the Quantity available : ";
    cin>>n;
    cout<<"Enter the cost : ";
    cin>>c;
    x.getdata(n,c);
    product y;
    cout<<"Enter the details of product Y \n";
    cout<<"Enter the Quantity available : ";
    cin>>n;
    cout<<"Enter the cost : ";
    cin>>c;
    y.getdata(n,c);
    cout<<"PRODUCT  AVAILABLE  PRICE \n";
    x.putdata('X');
    y.putdata('Y');
    return 0;
}
```

Output

```
Enter the details of product X
Enter the Quantity available : 70
Enter the cost : 94.50
Enter the details of product Y
Enter the Quantity available : 50
Enter the cost : 23.25
PRODUCT  AVAILABLE  PRICE
X        70              94.5
Y        50              23.25

--------------------------------
Process exited after 20.71 seconds with return value 0
Press any key to continue . . .
```

21CSC101T - OBJECT ORIENTED DESIGN AND PROGRAMMING

Session 8

Topic : Feature Abstraction and Encapsulation, Application of Abstraction and Encapsulation

Abstraction

- Data abstraction allows a program to ignore the details of how a data type is represented.

- Abstraction (derived from a Latin word **abs**, meaning away from and **trahere**, meaning to draw) refers to the act of representing essential features without including the background details or explanations.

- C++ classes use the technique of abstraction and are defined as a list of abstract attributes such as width, cost, size etc and functions to operate on these attributes.

Abstraction

- While classifying a class, both data members and member functions are expressed in the code. But, while using an object (that is an instance of a class) the built-in data types and the members in the class get ignored which is known as data abstraction.

- C++ provides a great level of data abstraction. In C++, we use classes to define our own abstract data types (ADT). Programmers can use the "cout" object of class ostream for data streaming to standard output like this:

Abstraction - Example

Example:

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


int main()
{
    cout << "Hello World" << endl;
}
```

What is abstraction

- Abstract class in C++ is the one which is not used to create objects.
- These type of classes are designed only to treat like a base class (to be inherited by other classes).
- It is a designed technique for program development which allows making a base upon which other classes may be built.

Here is an example of declaring Public members of C++ class

```
#include <iostream>
using namespace std;
class implementAbstraction
{
   private:
      int a, b;
   public:
            void set(int x, int y)
      {
         a = x;
         b = y;
      }
      void display()
      {
         cout<<"a = " <<a << endl;
         cout<<"b = " << b << endl;
      }
};
```

```
int main()
{
   implementAbstraction obj;
   obj.set(10, 20);
   obj.display();
   return 0;
}
}
```

Output:

```
a = 10
b = 20
```

Here is a Private member example in which member data cannot be accessed outside the class

```
#include <iostream>
using namespace std;
class sample {
public:
   int gl, g2;
public:
   void val()
   {       cout << "Enter Two values : "; cin >> gl >> g2;
   }
private:
   void display()
   {       cout << gl << " " << g2;
      cout << endl;
   }
};
int main()
{
   sample S;
   S.val();
   S.display();
}
```

If you execute the above program, the private member function will not be accessible and hence the following error message will appear like this in some compiler:

Output:

```
error: 'void sample::display()' is private
     void display()
          ^
```

Advantages of Data Abstraction

- Class internals get protected from user-level errors
- Programmer does not have to write the low-level code
- Code duplication is avoided and so programmer does not have to go over again and again fairly common tasks every time to perform similar operation
- The main idea of abstraction is code reuse and proper partitioning across classes
- For small projects, this may not seem useful but for large projects, it provides conformity and structure as it provides documentation through the abstract class contract
- It allows internal implementation details to be changed without affecting the users of the abstraction

Data Encapsulation

- Unable to deal with the complexity of an object, human chooses to ignore its non-essential details and concentrate on the details which are essential to our understanding.
- Abstraction is "looking for what you want" within an object or class.
- But there is another major concept connected to abstraction which is called encapsulation.
- So basically, encapsulation can be defined as the process of hiding all of the details of an object that do not throw in or dealt with its essential characteristics.
- Encapsulation can also be defined as preventing access to non-essential details of classes or its objects.
- Abstraction and encapsulation have close bonding among each other.
- Encapsulation assists abstraction by providing a means of suppressing the non-essential details.

ENCAPSULATION

ADVANTAGES

- **Encapsulated classes reduce complexity.**
- **Help protect our data.** A client cannot change an Account's balance if we encapsulate it.
- **Encapsulated classes are easier to change.**

The function which we are making inside the class ,it must use the all member variable then only it is called encapsulation.

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

class Encapsulation
{
    private:
        // data hidden from outside world
        int x;

    public:
        // function to set value of
        // variable x
        void set(int a)
        {
            x =a;
        }

        // function to return value of
        // variable x
        int get()
        {
            return x;
        }
};
```

```
int main()
{
    Encapsulation obj;

    obj.set(5);

    cout<<obj.get();
    return 0;
}
```

output:

5

Abstraction	Encapsulation
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
4. **Abstraction**- Outer layout, used in terms of design. For Example:- Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number.	4. **Encapsulation**- Inner layout, used in terms of implementation. For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits.

Application of Abstraction

- For example, when you send an email to someone you just click send and you get the success message, what actually happens when you click send, how data is transmitted over network to the recipient is hidden from you

Example

- The common example of encapsulation is **Capsule**. In capsule all medicine are encapsulated in side capsule.

- **Automatic Cola Vending Machine :**Suppose you go to an automatic cola vending machine and request for a cola. The machine processes your request and gives the cola.

Examples

Data Abstraction and Encapsulation

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction

```
#include <iostream>
using namespace std;
class Adder {
  public:
    // constructor
    Adder(int i = 0) {
      total = i;
    }
     // interface to outside world
    void addNum(int number) {
      total += number;
    }

    // interface to outside world
    int getTotal() {
      return total;
    };
     private:
    // hidden data from outside world
    int total;
};
```

ACCESS SPECIFIERS

Session - 11

Access control in classes

One of the main features of object-oriented programming languages such as C++ is **data hiding**. Data hiding refers to restricting access to data members of a class. This is to prevent other functions and classes from tampering with the class data.

Definition :

The access specifiers of C++ allows us to determine which class members are accessible to other classes and functions, and which are not.

Types of C++ Access Specifiers are:

- **public :** A public member is accessible from anywhere outside the class but within a program.
- **private :** A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.
- **protected :** A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes

Syntax of Declaring Access Specifiers in C++

```
class ClassName
    {
            private:
                // declare private members/methods here
            public:
                // declare public members/methods here
            protected:
                // declare protected members/methods here
    };
```

Access Specifiers in C++

Specifier	Within Same Class	In Derived Class	Outside the Class
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

Example 1: C++ public Access Specifier

```
#include <iostream>
using namespace std;
class Sample                    // define a class
{
public:                         // public elements
    int age;
    void displayAge()
 {
        cout << "Age = " << age << endl;
    }
};
```

```
int main()
  {
    Sample obj1;     // declare a class object

    cout << "Enter your age: ";

    // store input in age of the obj1 object
    cin >> obj1.age;

  obj1.displayAge();              // call class function

    return 0;
}
```

Output:

Enter your age: 20
Age = 20

Example 2: C++ private Access Specifier

```
class Sample
 {
private:              // private elements
int age;

public:               // public elements
void displayAge(int a)
 {
age = a;
cout << "Age = " << age << endl;
}
 };
```

```
int main()
{
 int ageInput;
Sample obj1;
cin >> obj1.age;                          // error
cout << "Enter your age: ";
cin >> ageInput;
obj1.displayAge(ageInput);
return 0;
}
```

Output:

Enter your age: 20
Age = 20

Example 3 : C++ protected Access Specifier

```
class Sample          // declare parent class
{
  protected:          // protected elements
   int age;
};

// declare child class
class SampleChild : public Sample
 {
  public:
   void displayAge(int a)
   {
      age = a;
      cout << "Age = " << age << endl;
   }

};
```

```
int main()
{
   int ageInput;

   // declare object of child class
   SampleChild child;

   cout << "Enter your age: ";
   cin >> ageInput;

   // call child class function
   child.displayAge(ageInput);

   return 0;
}
```

Output:

```
Enter your age: 20
 Age = 20
```

Access Specifiers Example 1

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

class Line
{
   public:
     double length;
     void setLength( double len );
     double getLength( void );
};

 double Line::getLength(void)
{
    return length ;
}

void Line::setLength( double len )
{
    length = len;
}
```

```
// Main function for the program

int main( )
{
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   // set line length without member function

   line.length = 10.0; // OK: because length is public
   cout << "Length of line : " << line.length <<endl;
   return 0;
}
```

```
Length of line : 6
Length of line : 10
```

Access Specifiers Example 2

```
#include <iostream>

using namespace std;

class Box
{
   public:
      double length;
      void setWidth( double wid );
      double getWidth( void );

   private:
      double width;
};

// Member functions definitions
double Box::getWidth(void)
{
    return width ;
}

void Box::setWidth( double wid )
{
    width = wid;
}
```

```
// Main function for the program
int main( )
{
   Box box;

   // set box length without member function
   box.length = 10.0; // OK: because length is public
   cout << "Length of box : " << box.length <<endl;

   // set box width without member function
   // box.width = 10.0; // Error: because width is private
   box.setWidth(10.0); // Use member function to set it.
   cout << "Width of box : " << box.getWidth() <<endl;

   return 0;
}
```

```
Length of box : 10
Width of box : 10
```

Access Specifiers Example 3

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

class Box
{
   protected:
      double width;
};

class SmallBox:Box
{
   public:
      void setSmallWidth( double wid );
      double getSmallWidth( void );
};

// Member functions of child class
double SmallBox::getSmallWidth(void)
{
    return width ;
}
```

```
void SmallBox::setSmallWidth( double wid )
{
    width = wid;
}

// Main function for the program
int main( )
{
   SmallBox box;

   // set box width using member function
   box.setSmallWidth(5.0);
   cout << "Width of box : "<< box.getSmallWidth();

   return 0;
}
```

```
Width of box : 5
```

Friend function

- A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.

- Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

- A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

Friend function Example

```
#include <iostream>
 using namespace std;
 class XYZ
 {
 private: int num=100;
 char ch='Z';
 public:
friend void disp(XYZ obj);

 };
```

```
//Global Function
void disp(XYZ obj)
{

cout<<obj.num<<endl;
cout<<obj.ch<<endl;
}
int main()
{
XYZ obj;
disp(obj);
return 0;
 }
```

C++ friend function used to print the length of the box.

```
#include <iostream>
using namespace std;
class Box
{
  private:
      int length;
  public:
       friend int printLength (Box);       //friend function
};
int printLength (Box b)
{
   b. length +=10;
   return b. length;
}
int main ()
{
  Box b;
  cout <<" Length of box:" <<printLength (b)<<endl;
   return 0;
}
```

Simple example when the function is friendly for two classes

```
#include<iostream>
using namespace std;
class B; //forward declaration.
class A
{
   int x;
   public:
       void setdata (int i)
        {
          x=i;
        }
   friend void max (A, B); //friend function.
} ;
class B
{
    int y;
    public:
        void setdata (int i)
         {
           y=i;
         }
    friend void max (A, B);
};
```

```
void max (A a, B b)
{
  if (a.x >= b.y)
       std:: cout<< a.x << std::endl;
  else
       std::cout<< b.y << std::endl;
}
 int main ()
{
  A a;
  B b;
   a. setdata (10);
   b. setdata (20);
   max (a, b);
   return 0;
}
```

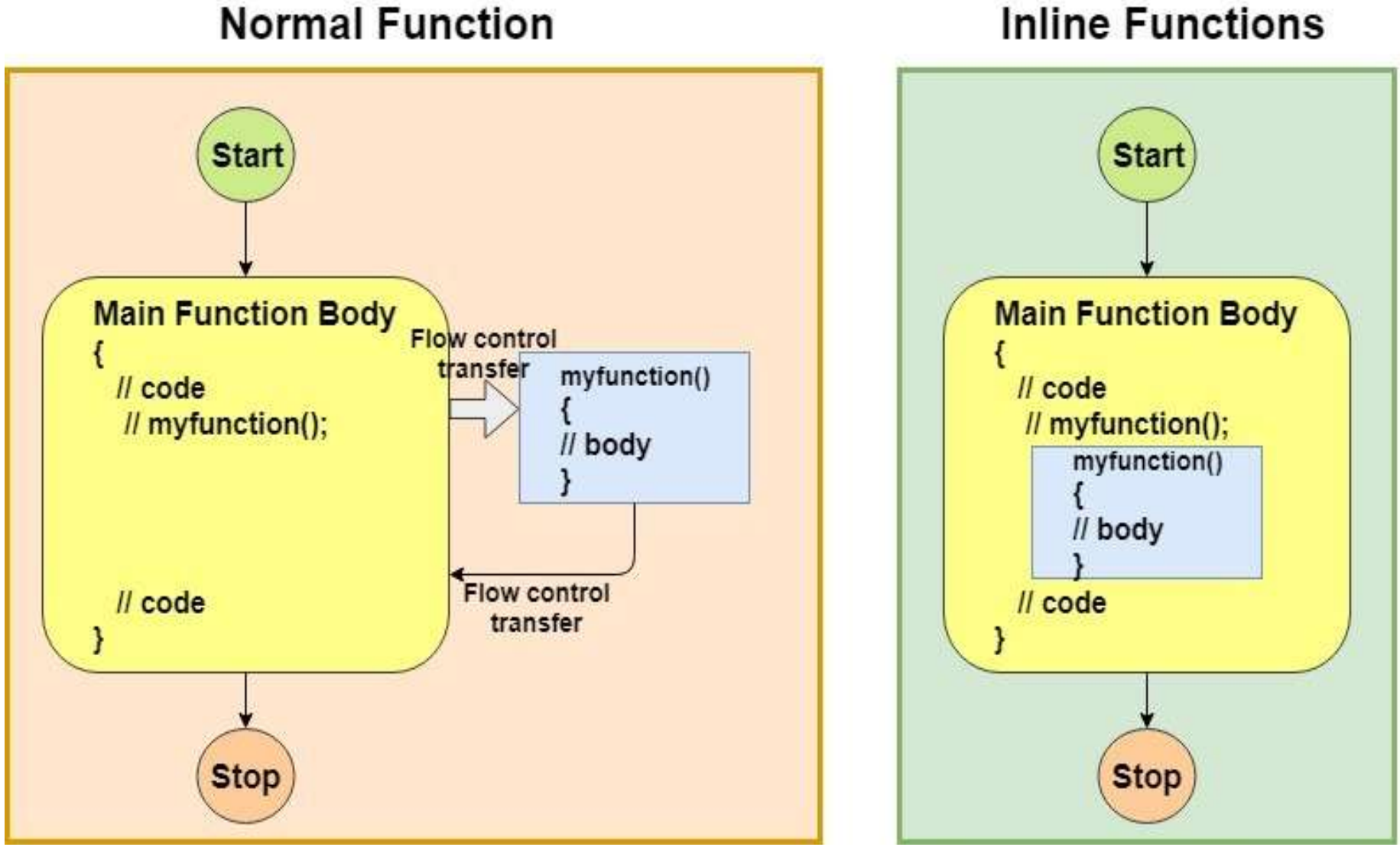
A simple example of a friend class:

```
#include <iostream>
using namespace std;
class A
{
  int x=4;
  friend class B; //friend class
};
class B
{
  public:
  void display (A &a)
   {
     cout<<"value of x is:" <<a.x;
   }
};
int main ()
{
  A a;
  B b;
  b. display (a);
   return 0;
}
```

Inline Function

- C++ provides an inline functions to reduce the function call overhead.
- Inline function is a function that is expanded in line when it is called.
- When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time.
- Inline function may increase efficiency if it is small.

Inline Function - Example



Inline Function - Example

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
   return s*s*s;
}
int main()
{
   cout << "The cube of 3 is: " << cube(3) << "\n";
   return 0;
}

//Output: The cube of 3 is: 27
```

Inline Function - Example

```
include <iostream>
using namespace std;
inline int Max(int x, int y) {
return (x > y)? x : y;
 }
// Main function for the program
 int main() {
cout << "Max (20,10): " << Max(20,10) << endl;
cout << "Max (0,200): " << Max(0,200) << endl;
cout << "Max (100,1010): " << Max(100,1010) << endl;
 return 0;
 }
```

Output:

Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010

Session 12

Topic : UML use case Diagram, use case, Scenario, Use case Diagram objects and relations

USECASE DIAGRAM

- Use case diagrams give us that capability.
- Use case diagrams are used to depict the context of the system to be built and the functionality provided by that system.
- They depict who (or what) interacts with the system. They show what the outside world wants the system to do.

The purposes of use case diagrams can be said to be as follows

- Used to gather the requirements of a system.
- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.
- Show the interaction among the requirements are actors.

NOTATIONS

- Actors are entities that interface with the system.
- They can be people or other systems.
- Actors, which are external to the system they are using, are depicted as stylized stick figures.

Figure 5–20 Actors

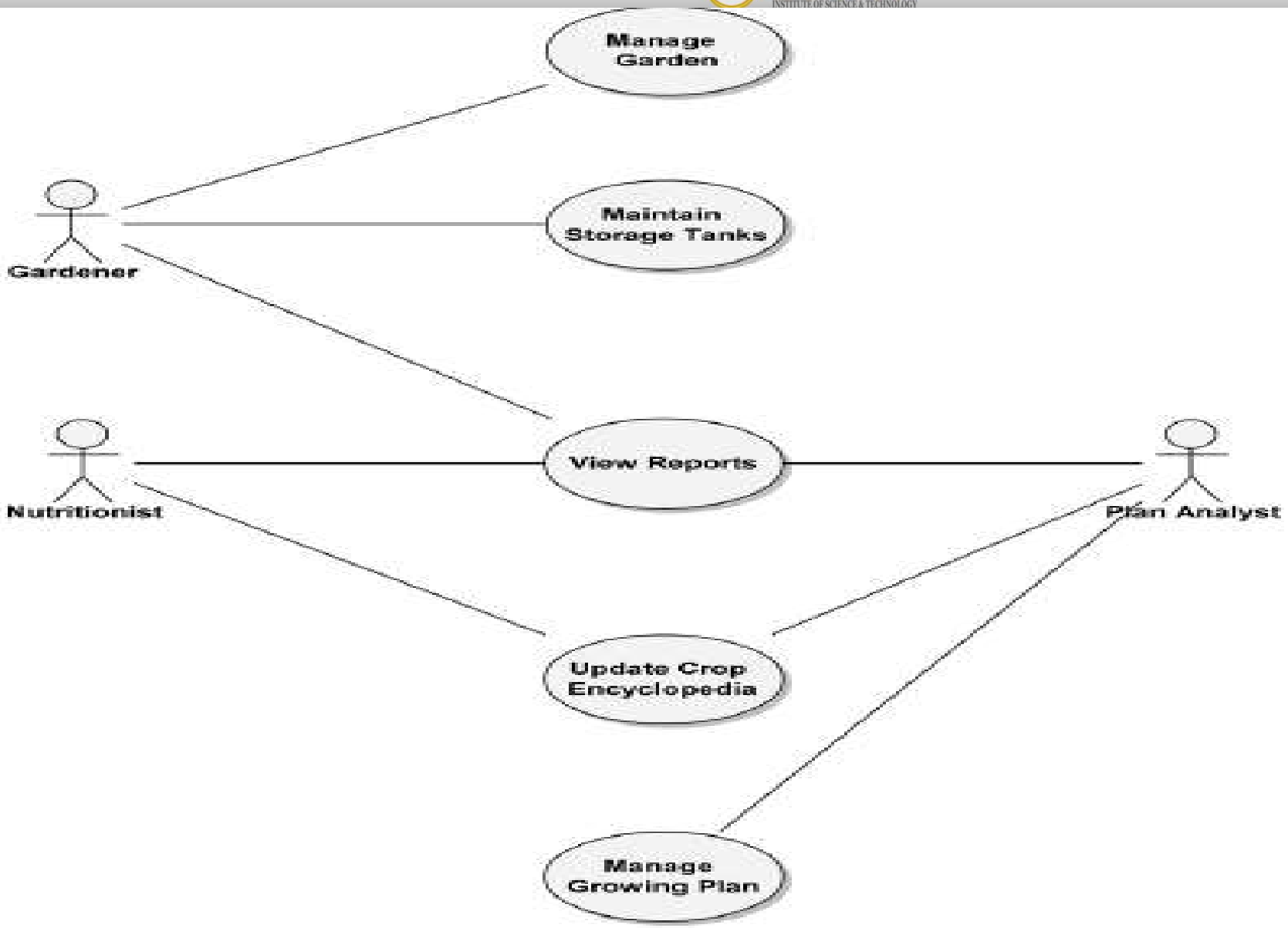


Figure 5–22 A Use Case Diagram

An Example Use Case Specification

Let us look at an example for the use case `Maintain Storage Tanks`.

Use Case Specification

Use case name: `Maintain Storage Tanks`

Use case purpose: This use case provides the ability to maintain the fill levels of the contents of the storage tanks. This use case allows the actor to maintain specific sets of water and nutrient tanks.

Optimistic flow:

A. Actor examines the levels of the storage tanks' contents.

B. Actor determines that tanks need to be refilled.

C. Normal hydroponics system operation of storage tanks is suspended by the actor.

D. Actor selects tanks and sets fill levels.

For each selected tank, steps E through G are performed.

E. If tank is heated, the system disables heaters.

1. Heaters reach safe temperature.

F. The system fills tank.

G. When tank is filled, if tank is heated, the system enables heaters.

1. Tank contents reach operating temperature.

H. Actor resumes normal hydroponics system operation.

Pragmatic flows:

Conditions triggering alternate flow:

Condition 1: There is insufficient material to fill tanks to the levels specified by the actor.

D1. Alert actor regarding insufficient material available to meet tank setting. Show amount of material available.

D2. Prompt actor to choose to end maintenance or reset fill levels.

D3. If reset chosen, perform step D.

D4. If end maintenance chosen, perform step H.

D5. Else, perform step D2.

Condition 2: . . .

Relationship

Two relationships used primarily for organizing use case models are both powerful

- «include» relationship
- «extend» relationship

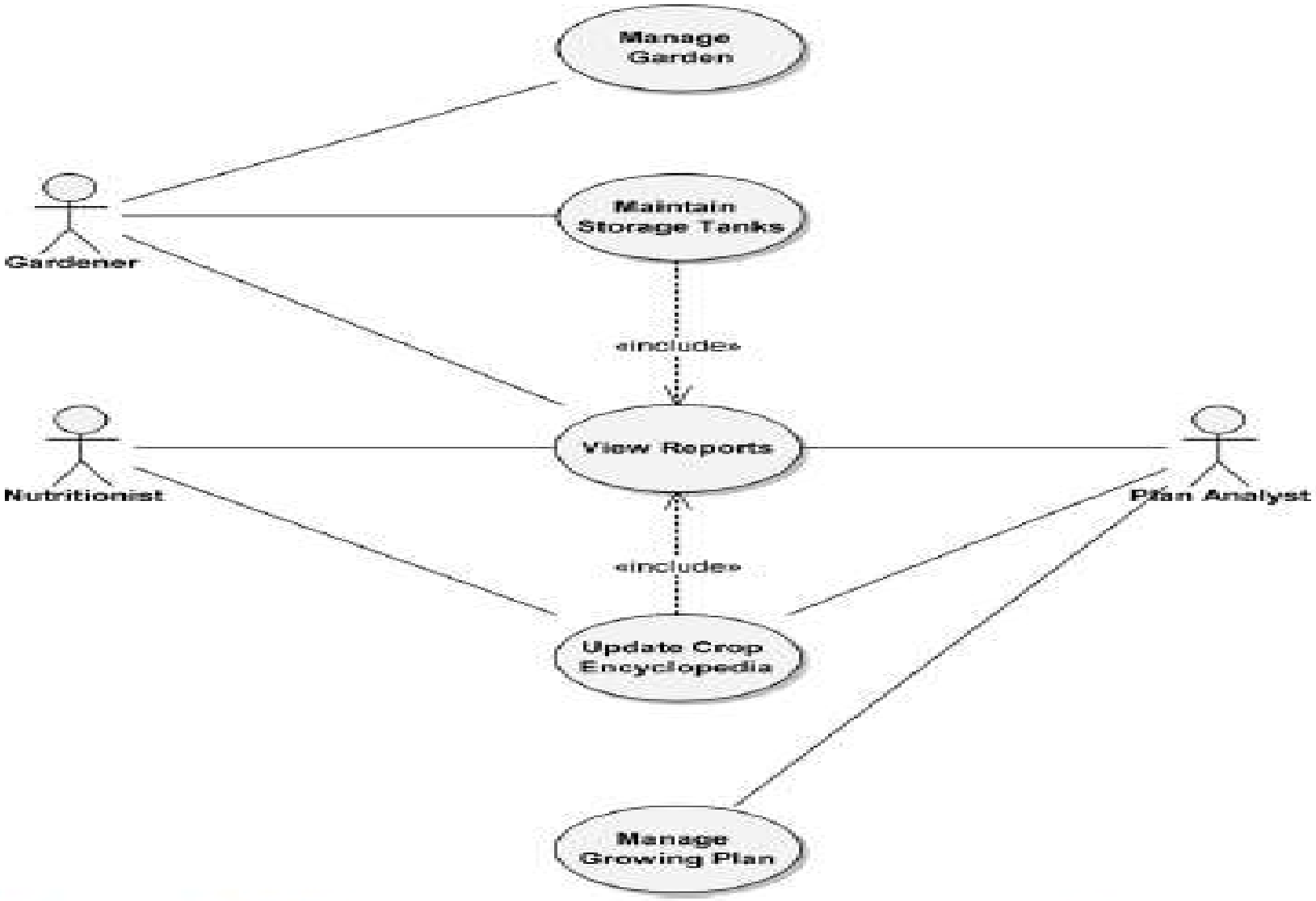


Figure 5–23 A Use Case Diagram Showing «include» Relationships

«include» relationship

- In our hydroponics example, we have an *Update Crop Encyclopedia* use case.
- During analysis, we determine that the *Nutritionist* actor using that use case will have to see what is in the crop encyclopedia prior to updating it.
- This is why the *Nutritionist* can invoke the View Reports use case.
- The same is true for the *Gardener* actor whenever invoking *Maintain StorageTanks*.

- Neither actor should be executing the use cases blindly. Therefore, the *View Report* use case is a common functionality that both other use cases need.
- This can be depicted on the use case model via an *«include» relationship*, as shown in Figure.
- This diagram states, for example, that the *Update Crop Encyclopedia* usecase includes the View Reports use case.
- This means that *View Reports* must be executed when *Update Crop Encyclopedia* is executed.
- *UpdateCrop Encyclopedia* would not be considered complete without View Reports.

«extend» Relationships

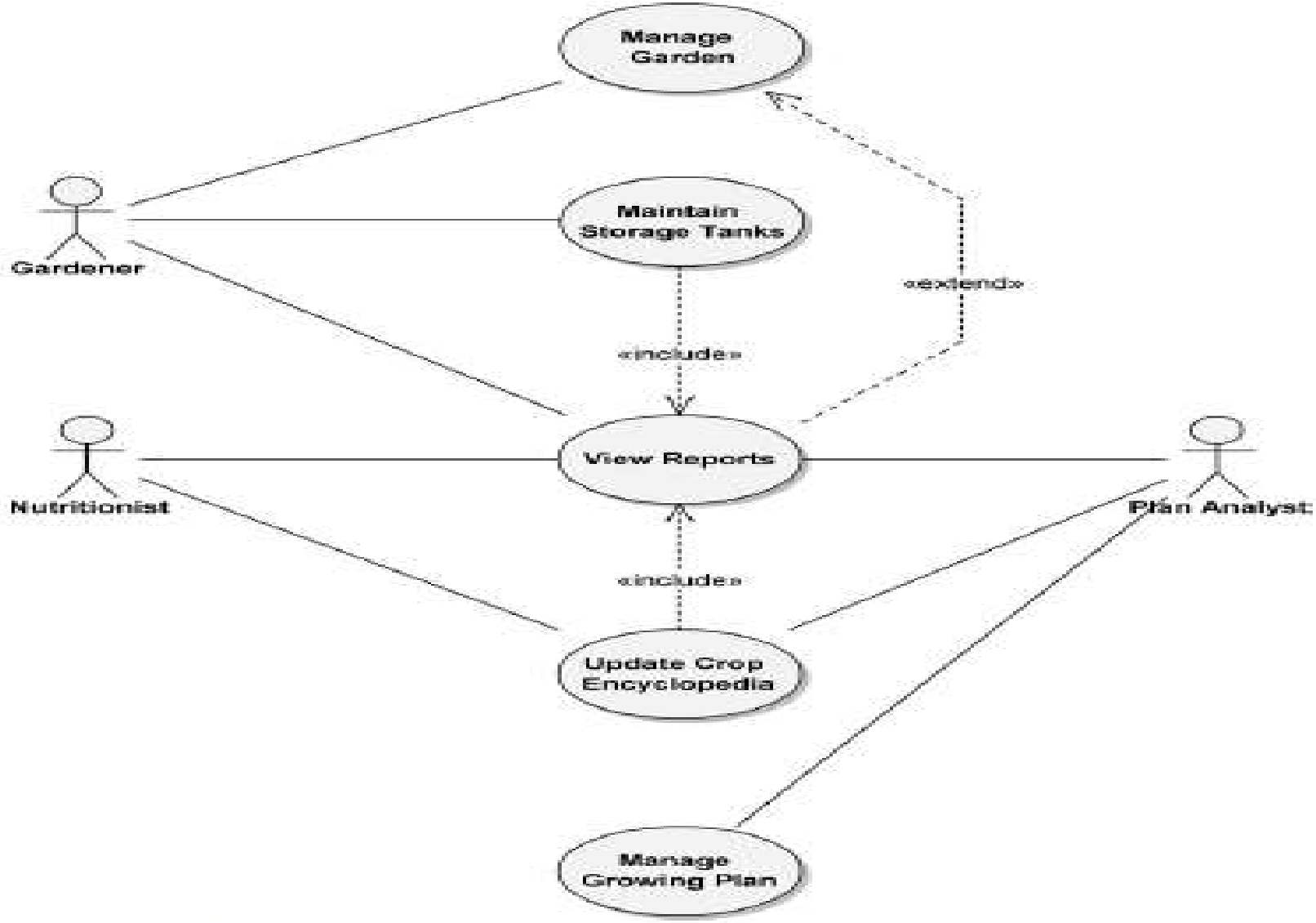


Figure 5–24 A Use Case Diagram Showing an <<extend>> Relationship

- While developing your use cases, you may find that certain activities might be performed as part of the use case but are not mandatory for that use case to run successfully.
- In our example, as the *Gardener* actor executes the *Manage Garden* use case, he or she may want to look at some reports.
- This could be done by using the *View Reports* use case.
- However, *View Reports* is not required when *Manage Garden* is run. *Manage Garden* is complete in and of itself. So, we modify the use case diagram to indicate that the *View Reports* use case extends the *Manage Garden* use case.

- Where an extending use case is executed, it is indicated in the use case specification as an extension point.
- The extension point specifies where, in the flow of the including use case, the extending use case is to be executed.

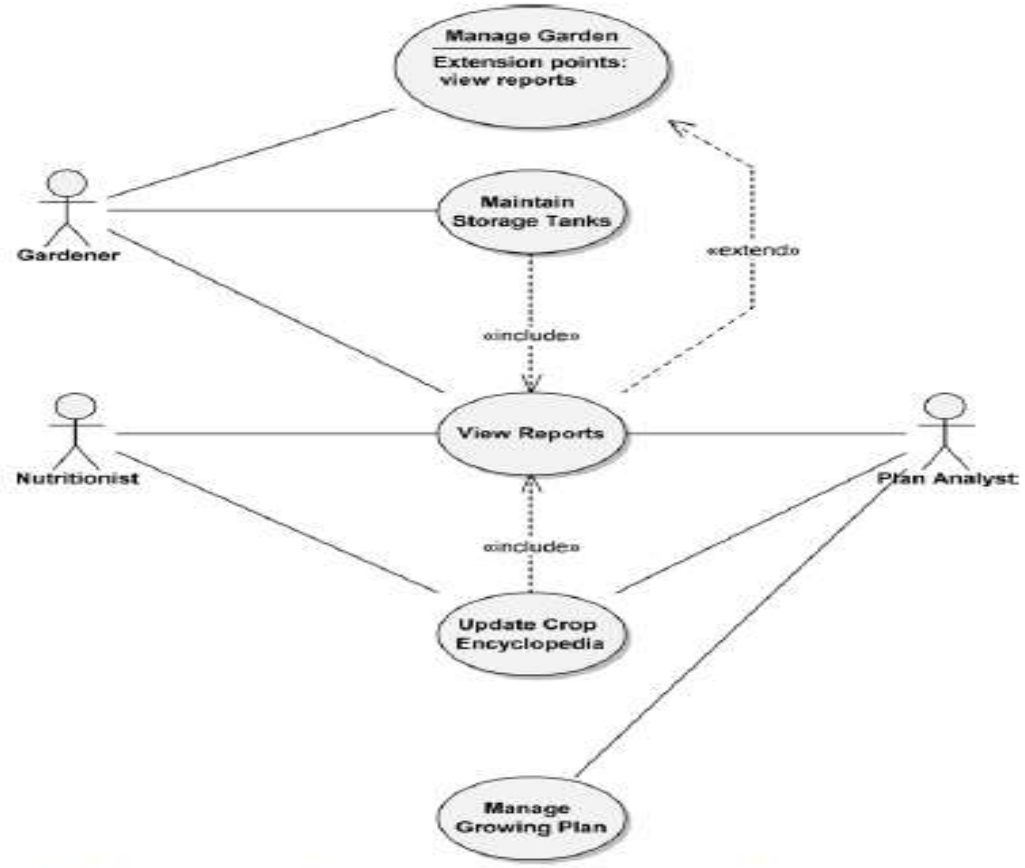


Figure 5–25 A Use Case Diagram Showing an Extension Point

Table 5–1 Key Differences between «include» and «extend» Relationships[a]

	Included Use Case	Extending Use Case
Is this use case optional?	No	Yes
Is the base use case complete without this use case?	No	Yes
Is the execution of this use case conditional?	No	Yes
Does this use case change the behavior of the base use case?	No	Yes

Session 13

Topic : Constructor and Destructor

- **Constructor is a special type of member function which is primarily used to initialize the member variables value.**
- **Constructor get automatically invoked** when instance or **object for the class is created**
- When a class is instantiated, even if we don't declare a constructor, compiler automatically creates one for the program. This compiler created constructor is called default constructor.

Rule for defining constructor:

- The name of the constructor must be the same as the name of the class.
- Constructor don't have any return type, always of type void.
- A Class can have any number of constructor provided properly overloaded.

Features of Constructor

- It should be declared in public scope.
- It is **invoked automatically** whenever an object is created.
- It **doesn't have any return type,** not even void. Hence, it can't return values.
- It **can't be inherited,** though a derived class can call the base class constructor.
- Constructors should always be non-virtual. They are static functions and in C++ so they cannot be virtual.
- A **constructor is executed repeatedly** whenever the objects of a class are created.
- A class can have more than one constructor i.e. constructors can be overloaded.

Syntax for Constructor:

```
class classname
{
public:
        classname ([parameter_list])
        {
                //constructor definition
        }
};
```

Example:

```
class Student
{
        int rno;
public:
        Student(int no)
        {
                rno = no;
        }
};
```

Types of Constructor

Types of Constructors in C++

Constructors are of three types:

- **Default Constructor -** Default constructor is the constructor which doesn't take any argument. It has no parameter.
- **Parametrized Constructor :** These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.
- **Copy Constructor:** These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object.

Default Constructor:

- A constructor that accepts no parameters is called Default Constructor.
- Compiler supplies a default constructor if no such constructor is defined.
- In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 or any random integer value in this case.

Example:

```
class Bank_Account
{
        int accno,balance;
public:
        Bank_Account()
        {
                accno =9999;
                balance=1000;
        }
};
```

```
int main()
{
        Bank_Account b1;
        return 0;
}
```

Parameterized Constructor:

The Constructors that can take arguments are called parameterized constructor. Sometimes, it may be necessary to initialize the data members of different objects with different values when they are created.

Example:

```
class Bank_Account
{
        int accno,balance;
public:
        Bank_Account(int a, int b)
        {
                accno =a;
                balance=b;
        }
};
int main()
{
        Bank_Account b1(101,10000);
        return 0;
}
```

Example for constructor with parameter

```
#include <iostream>
using namespace std;
class Employee {
  public:
      int id;
      string name;
      float salary;
      Employee(int i, string n, float s)
       {
          id = i;
          name = n;
          salary = s;
       }
      void display()
       {
          cout<<id<<"  "<<name<<"
"<<salary<<endl;
       }
};
```

Creating an object of Employee

```
int main(void) {
    Employee e1 =Employee(101, "Sonoo",
890000);
Employee  e2=Employee(102, "Nakul",
59000);
    e1.display();
    e2.display();
    return 0;
}
```

Copy Constructor

Copy constructor is used for creating a new object as a copy of an existing object. It si used when we need to coy data members from one object to another object. It takes reference to an object of the same class as an argument.

Copy constructors are called using the cases below:

- Whenever objects of the class are returned by value.
- Whenever objects are constructed based on another object that is of the same class.
- Whenever objects are passed as parameters.

Copy Constructor

Example:

```
class integer
{
        int a ,b;
        public:
                // copy constructor declared
               integer (integer &v)
               {
                        a=v.a;
                        b=v.b;
               };
};
```

```
int main()
{
        integer ob1;
        integer ob2(ob1);
        return 0;
}
```

Example for copy constructor

```
#include <iostream>
using namespace std;
class A
{
  public:
   int x;
   A(int a)    // parameterized constructor.
   {
     x=a;
   }
   A(A &i)             // copy constructor
   {
       x = i.x;
   }
};
```

```
int main()
{
  A a1(20);              // Calling the
parameterized constructor.
 A a2(a1);              //  Calling the
copy constructor.
 cout<<a2.x;
  return 0;
}
```

Output

20

Multiple Constructors

Example:

```
class Student
{
        public:
                int rollno;
                string name;
                // first constructor
                Student(int x, string str)
                {
                        rollno = x;
                        name = str;
                }
                //second constructor
                Student( Student &s1)
                {
                        rollno= s1.rollno;
                        name=s1.name;
                }
void display()
{
cout<<this->rollno<<“ "<<this-> name <<endl;
}
};
int main()
{
        Student s1(101,”Abinav”);
        Student s2(s1);
        s1.display();
        s2.display();
        return 0;
}
```

Output:

```
101 Abinav
101 Abinav
```

Destructor

- Destructor is a special class function **which destroys the object as soon as the scope of object ends**. The destructor is called automatically by the compiler when the object goes out of scope.
- 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.

Syntax:

```
Class   classname
{
        public      ~classname()
        {}
};
```

When does the destructor get called?

A destructor is **automatically called** when:

1) The program finished execution.

2) When a scope (the { } parenthesis) containing **local variable** ends.

3) When you call the delete operator.

Features of Destructor

- When objects are destroyed, the destructor function is automatically called.
- It's not possible to declare it static or const.
- There are no arguments for the destructor.
- It doesn't even have a void return form.
- In the public section of the class, a destructor should be declared.
- Destructors cannot be overloaded i.e., there can be only one destructor in a class
- They cant be inherited.

Destructors Example:

```
class A
{
        // constructor
         A()
        {
              cout << "Constructor called";
        }

        // destructor
        ~A()
        {
                    cout << "Destructor called";
        }
};
```

```
int main()
{
        A obj1;   // Constructor Called
        int x = 1
        if(x)
        {
                    A obj2;  // Constructor Called
        }   // Destructor Called for obj2
} //  Destructor called for obj1
```

Example:

```
#include <iostream>
using namespace std;
class Student
{
        public:
                int rollno;
                string name;
        //default Constructor
         Student()
        {
                rollno=-1;
                name="";
        }
        // parametrized constructor
        Student(int x, string str)
        {
                rollno = x;
                name = str;
        }
Student( Student &s1)
{
        rollno= s1.rollno;
        name=s1.name;
}
void display()
{
cout<<this->rollno<<" "<<this->name<<endl;
}
//destrcutor
~Student()
{
        cout<<"destructor invoked"<<endl;
}
};
int main() {
        Student s;
        Student s1(101,"Abinav");
        Student s2(s1);
        s.display();
         s1.display();
        s2.display();
        return 0;
}
```

Output

```
-1
101 Abinav
101 Abinav
destructor invoked
destructor invoked
destructor invoked
```

Methods

- A *method* is a function that is part of the class definition. The methods of a class specify how its objects will respond to any particular message.
- A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything.
- Methods allow us to reuse the code without retyping the code.
- A *message* is a request, sent to an object, that activates a method (i.e., a member function).

Method Vs Constructor

- Used to define particular task for execution
- Method can have same name as class name or different name based on the requirement
- Explicit call statement required to call a method
- There is no default method provided by compiler
- Return data type must be declared

- Used to initialize the data members
- Constructor has same name as the class name
- Constructor is automatically called when an object is created
- There is always default constructor provide by compiler
- Return type is not required in constructor


Thank
you