18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

Unit 1

Introduction

Object Oriented Programming

- **OOP** is a programming paradigm / model that organizes software design around data or objects rather than functions
- **Object** is a basic unit of OOP, It can be defined as data field that has unique attributes and behavior.
- Aims to implement real world entities like inheritance, hiding, polymorphism etc in programming
- Bind data and functions together
- Functions associated with the data object can only access this data, No other part of code can access
- Code reusable - easily maintain and modify existing code

Object Oriented Design

- It is the process of using an object oriented methodology to design a computing system or application
- Provides a system of interacting objects for the purpose of solving a software problem

Comparison of Procedural and Object-Oriented Programming

S.No	Procedure Oriented/ Structure Programming	Object Oriented Programming
1.	It follows top down approach.	It follows bottom up approach.
2.	Program is divided into small parts called functions	Program is divided into small parts called objects.
3.	function is more important than data.	data is more important than function
4.	There is no access specifier in procedural programming.	have access specifiers like private, public, protected etc.
5.	It is less secure than OOPs. Because no data hiding.	Data hiding is possible, hence more secure than procedural.
6.	Data moves freely within the system from one function to another.	In OOP, objects can move and communicate with each other via member functions.

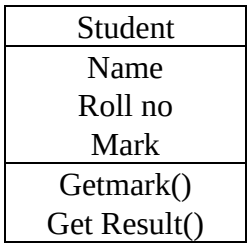
7.	overloading is not possible.	Overloading is possible in object oriented programming.
8.	No inheritance.	inheritance is present in object-oriented programming.
9.	There is no code reusability. Adding new data and function in POP is not so easy.	It offers code reusability by using the feature of inheritance. OOP provides an easy way to add new data and function.
10.	Does not Supports Polymorphism	Supports Polymorphism
11.	Does not Supports Virtual Functions	Supports Virtual Functions
12.	Examples: C, FORTRAN, Pascal, Basic etc.	Examples: C++, Java, Python, C# etc.

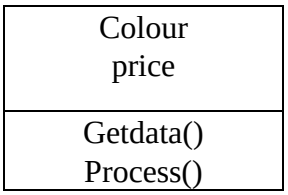
OOPS and its features

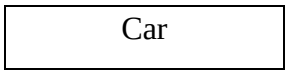
1. Objects
2. Classes
3. Encapsulation
4. Data abstraction
5. Polymorphism
6. Inheritance
7. Dynamic binding
8. Message passing

1. Objects:

- Basic unit of Oops
- Instant of a class
- Run time entity
- Has object name, data(attributes), Methods(functions)



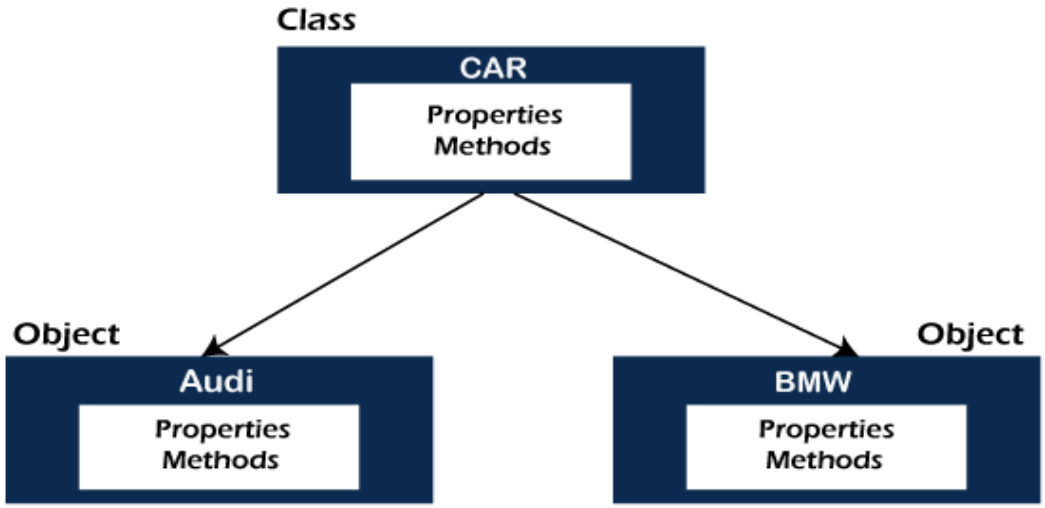




2. Class

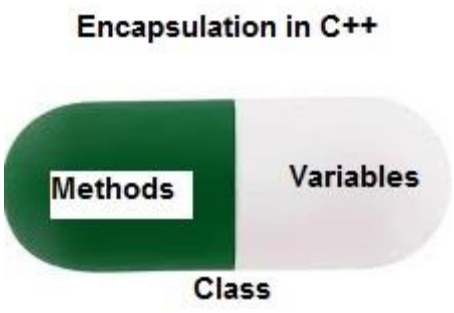
- Template or blueprint that describes the object
- Has class name, attributes (data members), Methods
- Memory is allocated at run time when object created

Student
Name Roll no Mark
Getmark() Get Result()



3. Encapsulation:

- Wrapping of data and functions together as a single unit is known as encapsulation.
- By default data is not accessible to outside world and they are only accessible through the functions which are wrapped in a class.
- prevention of data direct access by the program is called data hiding or information hiding



4. Data abstraction:

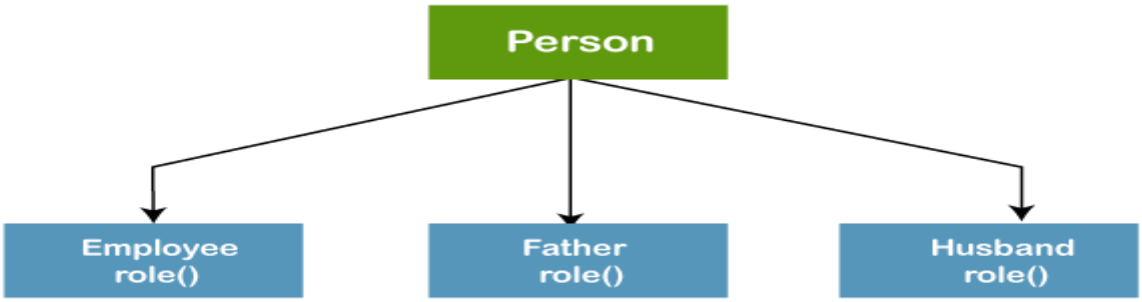
Data abstraction provides /exposes only essential features (like interface details) and hides background details (Implementation details).

Class use the concept of data abstraction so they are called **abstract data type** (ADT)

5.Polymorphism:

Polymorphism means the ability to take more than one form. For example, an operation have different behavior in different instances. The behavior depends upon the type of the data used in the operation.

Polymorphism comes from the Greek words “poly” and “morphism”. “poly” means many and “morphism” means form i.e.. many forms.



6.Inheritance:

Inheritance is the process of creating new classes by deriving the properties and characteristics from another existin class.

- It helps realize the goal of constructing software from reusable parts, rather than hand coding every system from scratch.
- When the class child, inherits the class parent, the class child is referred to as derived class (sub class) and the class parent as a base class (super class).
- In this case, the class child has two parts:
  - a derived part and an incremental part.
  - The derived part is inherited from the class parent.
  - The incremental part is the new code written specifically for the class child.

7. Dynamic binding:

Binding refers to linking of procedure call to the code to be executed at the run time.

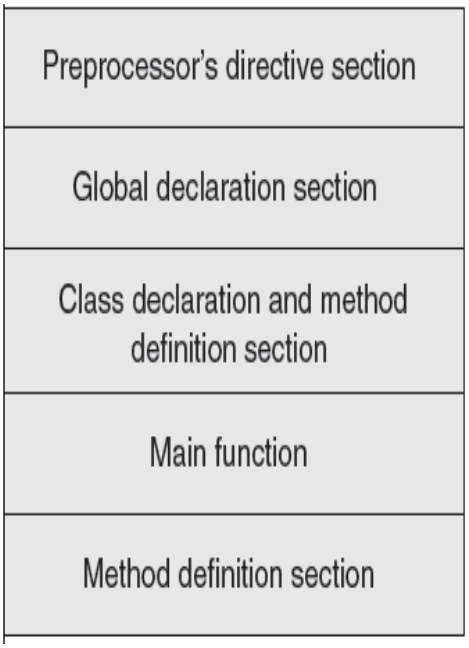
8. Message passing:

Objects communicates with each other by sending and receiving information.

C++ is a general purpose object programming language developed by Bjarne Stroustrup in 1980s at Bell Labs.

C++ is a superset of c, i.e. All C programs are also C++ programs

BASIC STRUCTURE OF C++ LANGUAGE



SIMPLE C++ PROGRAM 1:

```
#include <iostream>

using namespace std;

int main()
{
  cout << "Hello World" ;

  return 0;
 }
```

SIMPLE C++ PROGRAM 2:

```
#include <iostream>
```

```
using namespace std;

#include<iostream>
class student
{
private:
          int rollno;
          char name[25];
public:
       int getdata()
       {
         cout<<"Enter Rollno\n";
          cin>>rollno;
          cout<<"Enter Name\n";
            cin>>name;
            return 0;
          }


int display()
       { cout<<"Rollno="<< rollno<<"\n";
          cout<<"Name="<< name<<"\n";
          return 0;
        }
};

int main()
{
student s1;
cout<<"student1 Details\n";
s1.getdata();
s1.display();
return 0;
}
```

using namespace std;

namespace is used to define a scope that could hold global identifiers. Namespace is a feature added in C++ and not present in C. **Namespace std** contains all the classes, objects and functions of the standard C++ library.

Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope

I/O Operations, Data Types, Variables, Static

I/O Operations:

- C++ I/O operation uses the stream concept.
- Stream is the sequence of bytes or flow of data. It makes the performance fast.
- If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as output operation.
- If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as input operation.

Header File

<iostream> It is used to define the **cout, cin and cerr** objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.

Standard output stream (cout)

- The cout is a predefined object of output stream (ostream class).
- It is connected with the standard output device, which is usually a display screen.
- The cout is used in conjunction with stream **insertion operator (<<)** to display the output on a console

Example: cout<<"Enter Rollno";

Cout<<"Enter Rollno"<<Enter Name";

Standard input stream (cin)

- The cin is a predefined object of input stream (istream class).
- It is connected with the standard input device, which is usually a keyboard.
- The cin is used in conjunction with stream **extraction operator (>>)** to read the input from a console.

Example: cin>>name;

Cin>>rollno>>name;

Standard end line (endl)

- The endl is a predefined object of ostream class.
- It is used to insert a new line characters and flushes the stream.

Example: cout << " C++ Program"<<endl;

Note: We can use the << or >> operator multiple times in the same line. This is called cascading.

DATA TYPES:

A data type is used to indicate the **type of data value stored** in a variable.

Types of data types:

1. Primary (basic or fundamental) data types.

2. Derived data types.

3. User-defined data types

Primitive Data Types:

These data types are built-in or predefined data types and can be used directly by the user to declare variables. Primitive data types includes:

1. Integer (Keyword: int)
2. Character
3. Boolean
4. Floating Point
5. Double Floating Point
6. Valueless or Void
7. Wide Character

Primary data type	Description	Size
int	used to store whole numbers	4 bytes
Char	A single character can be defined as a character data type ndicated with single quotes (’a’, ’3’).	1 byte
Float	Floating point number represents a real number with 6 digits precision	4 bytes
Double	Floating point number represents 14 digits of precision.	8 bytes
bool	Boolean or logical data type is a data type, having two values (usually denoted true and false),	1 byte
Void	Void data type has no values. This is usually used to specify the return type of functions. The type of the function said to be void when it does not return any value to the calling function. This is also used for declaring general purpose pointer called void pointer	

wchar_t	Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by **wchar_t**.	2 or 4 bytes long.

Derived data types

The data-types that are derived from the primitive or built-in data types. Types are:

1. Arrays
2. Pointer
3. Functions
4. References

User-defined data types

The data types defined by the user are known as the user-defined data types. They are

1. Structure
2. Union
3. Class
4. Enumeration
5. Typedef

Variables

A named memory location is called variable. The name of a variable is an identifier token. When using a variable, we actually refer to address of the memory where the data is stored

Identifiers may contain numbers, letters, and underscores(_), and may not start with a number.

Rules for declaring Variables names:

- The first character must be an alphabet or underscore.
- It must consist of only letters, digits and underscore.
- Identifiers may have any length but only first 31 characters are significant.
- It must not contain white space or blank space.
- We should not use keywords as identifiers.
- Upper and lower case letters are different.
- Variable names must be unique in the given scope

Declaring Variables

Each variable to be used in the program must be declared. To declare a variable, specify the data type of the variable, followed by its name.

type variable_list;

Example:

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

Variables are declared at three basic places	**Location**
local variable	a variable is declared inside a function
formal parameter	a variable is declared in the definition of function parameters
global variable	variable is declared outside all functions

Initializing Variables

Static Initialization: Here, the variable is assigned a value in advance. This variable then acts as a constant.

Example:

Declaring the variable and then initializing it

```
int a;
a = 5;
```

Declaring and Initializing the variable together

```
int a = 5;
```

Declaring multiple variables simultaneously and then initializing them separately

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

Declaring multiple variables simultaneously and then initializing them simultaneously

```
int a=5, b = 10, c = 20;
```

Dynamic Initialization: Here, the variable is assigned a value at the run time. The value of this variable can be altered every time the program is being run.

int a;

cout << "Enter the value of a"; cin >> a;

Static

Static keyword has different meanings when used with different types. We can use static keyword with:

Static Variables: Variables in a method or function, Variables in a class
Static Members of Class : Class objects and Functions in a class

A static variable:

- When a variable is declared as static, space for it gets allocated for the lifetime of the program.
- Even if the function is called multiple times, space for the static variable is allocated only once
- The value of variable in the previous call gets carried through the next function call

Ex: `static int count = 0;`

Static Class objects and static member functions: Just like variables, objects and methods also when declared as static have a scope till the lifetime of program.

Static member functions: are allowed to access only the static data members or other static member functions**, they can not access the non-static data members or member functions of the class.**

Example Program:

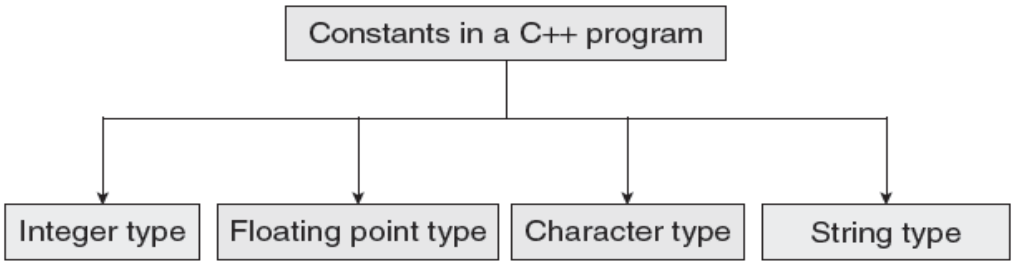
```
#include <iostream>
using namespace std;
  void demo()
{     // static variable
    static int count = 0;
    cout << count << " ";

    // value is updated and will be carried to next function calls
    count++;
}


int main()
{
```

```
    for (int i=0; i<5; i++)
        demo();
    return 0;
}
```

Constants, Pointers, Type Conversions

Constants:

- Constants are identifiers whose value does not change.
- While variables can change their value at any time, constants can never change their value.
- The value of the constant is known to the compiler at the compile time



Declaring a constant

- Constant names are usually written in capital letters to visually distinguish them from other variable.
- Can be defined using a preprocessor directive
- #define is a preprocessor compiler directive and not a statement. Therefore, it does not end with a semi-colon.
- Syntax: Const data_type const_name = value;

Example: Const float pi =3.14;

#define PI 3.14159

Integer constants: An integer constant is an integer-valued number. It consists of sequence of digits. Integer constants can be written in three different number systems:

1.Decimal integer (base 10).

2.Octal integer (base 8).

3.Hexadecimal (base 16).

Floating point constants or Real constants : The numbers with fractional parts are called real constants.

Representation: These numbers can be represented in either decimal notation or exponent notation (scientific notation).
Decimal notation: 1234.56, 75.098, 0.0002, -0.00674 (valid notations)

Exponent or scientific notation: General form: Mantissa e exponent

Mantissa: It is a real number expressed in decimal notation or an integer notation.
Exponent: It is an integer number with an optional plus (+) or minus (-) sign.
E or e: The letter separating the mantissa and decimal part.

Ex: 0.00000000987 is equivalent to **9.87e-9**

Character constants:-
***Single character constants*:** It is character (or any symbol or digit) enclosed within single quotes.
Ex: 'a'. '1'. '*'
Every Character constants have integer values known as **ASCII** values. (ASCII stands for American Standard Code for Information Interchange)

String constants or string literal:
String constant is a *sequence of zero or more characters* enclosed by double quotes.
Example:
"MRCET" "12345" "*)(&%"

Pointers:

- Pointer is a derived data type.
- A pointer is a variable used to store the address of a memory cell.
- Pointer variable can point to another pointer(double pointer)

Pointer Declarations

```
datatype *ptr_var_name;
ex: int *p1;    // pointer to int
 char *p2;      //pointer to char;
```

Initialization of ptr:

Can be initialized (assigned) with the address of another variable using & (address of) operator.
Ex: int *p1, a;
P1=&a;
Note : both the pointer and pointee should be of same data type.

Generic pointer:

Pointers of type void are known as generic pointers. It can address of any variable.

Accessing a variable using pointer:

Variable can be accessed using the Pointer variable with dereference operator (*).

Example:

```
#include <iostream>
using namespace std;
int main()
{
int var1=11;
int var2=22;
int *ptr1, *ptr2;
ptr1 =&var1; //pointer points to var1
ptr2 =&var2;
cout<<*ptr1<<" "<<*ptr2;
cout<<&ptr1<<" "<<&ptr2;
return 0;
 }
```

Output : 11 22 0x7ffe76905210 0x7ffe76905208

Accessing an array using pointer:

- Arrays can be accessed using the pointer variables.
- Base address (starting address) of an array is assigned to a pointer variable.
- Pointer variable is incremented to access the subsequent array elements.

Example:

```
#include <iostream>
using namespace std;
#include <iostream>
using namespace std;
int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  int *ptr = &arr[0];
```

```
  cout<<"The values in the array are: ";
  for(int i = 0; i < 5; i++) {
    cout<< *ptr <<" ";
    ptr++;
  }
  return 0;
}
// Note: for sum of array sum = sum+ *ptr;
```

Accessing an array using pointer:

- Arrays can be accessed using the pointer variables.
- Base address (starting address) of an array is assigned to a pointer variable.
- Pointer variable is incremented to access the subsequent array elements.

Example: swapping two numbers

```
#include <iostream>
using namespace std;
void swap( int *a, int *b )
{       int t;
        t = *a;
        *a = *b;
        *b = t;   }
int main(){
        int num1, num2;
        cout << "Enter first number and second number" << endl;
        cin >> num1 >> num2;
        swap( &num1, &num2);  // pass the values using the pointer
        cout << "After  swapping"<<endl<<"First number = " << num1 << endl;
        cout << "Second number = " << num2 << endl;
        return 0;   }
```

This pointer:

- When a member function is called this pointer is automatically passed as an implicit argument.
- It is the pointer to the invoking object.
- Member function can identify the object that invoked the function. i.e member function can find the address of the object.
- This pointer can be used to access the data in the object it points to.
- Objects can be returned by reference using this pointer rather than creating a temporary object.

Note:

To understand 'this' pointer, it is important to know how objects look at functions and data members of a class.

- Each object gets its own copy of data members and all objects share a single copy of member functions.
- Now, The compiler supplies an implicit pointer along with the names of the functions as 'this'.

Example Program:

```
#include<iostream>

using namespace std;

 * local variable is same as a member's name, In such case if you want to assign the local variable value to the data members then you won't be able to do until unless you use this pointer, because the compiler won't know that you are referring to object's data members unless you use this pointer.  */

class Test
{
private:
  int x;
public:
  void setX (int x)
  {
     // The 'this' pointer is used to retrieve the object's x
     // hidden by the local variable 'x'
    this->x = x;
    //  x=x; / * use x=x; instead of this->x = x and check, output willbe zero */

  }


  void display() { cout << "x = " << x << endl; }
```

```
};

int main()
{
  Test obj,obj1;
  int x = 20; int y=30;
  obj.setX(x);    obj1.setX(y);
  obj.display();   obj1.display();
   return 0;
 }
```

O/p: x=20 x=30

References:

A special type of variable is called as enumerated or reference variable is used to provide alias (alternative name) for previously defined variable.

Syntax: datatype &ref_name = var_name;

Eg: float total=100;

float &sum =total;

cout<<sum<<total; // both sum and total refers the same object in memory.

S.NO	References	Pointers
1	No null reference	Pointers can be made null
2	Once a reference is initialized to an object it cannot be changed to refer another object	Pointers can be pointed to another object at any time
3	A reference must be initialized when it is created	Pointers can be initialized at any time.

Type Conversions:

- Type conversion or typecasting of variables refers to changing a variable of one data type into another.

Implicit Type Conversion:

- While type conversion is done implicitly, (automatic type conversion Done by the compiler on its own)
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

  Example : `int x = 10; char y = 'a';`
  // y implicitly converted to int. ASCII
  // value of 'a' is 97

```
x = x + y;

// x is implicitly converted to float
float z = x + 1.0;
```

conversion hierarchy
bool -> char -> short int -> int -> unsigned int -> long int -> unsigned long int -> float -> double -> long double

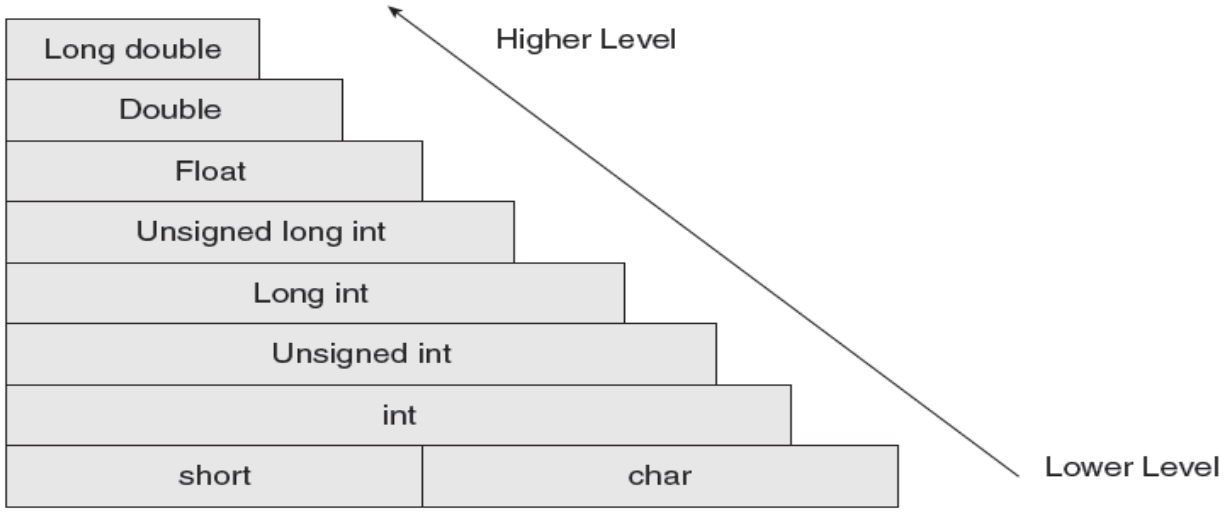


Figure 2.13 Conversion hierarchy of data types

Explicit Type conversion or casting

- This process is also called type casting and it is user-defined
- done explicitly by the programmer
- **Type casting an arithmetic expression** tells the compiler to represent the value of the expression in a certain format.
- **The general syntax for type casting is**
  destination_variable_name=destination_data_type(source_variable_name);

  Example 1:

```
float salary =10000.00;
int sal;
sal = (int)salary;
```

  Example 2:

```
double x = 1.2;
```

// Explicit conversion from double to int
int sum = (int) x + 1;

Example Program for type casting

```
#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;
}
```

Features: Class and Objects

Class:

- Template or blueprint that describes the object
- It is a collection of data and member functions that manipulate data.
- Memory is allocated at run time when object created
- It is also known as user defined data type or ADT(abstract data type)
- A class is declared by the keyword **class**.

Syntax:-

```
class class_name
{
        Access specifier :
                Variable declarations;
        Access specifier :
                function declarations;
};
```

Example of a class in C++:

```
class student
{
        private : int roll;
        char name[30];
        public:

        void get_data()
```

```
        {
        cout<<"Enter roll number and name":
        cin>>roll>>name;
        }

        void put_data()
        {
        cout<<"Roll number:"<<roll<<endl;
        cout<<"Name :"<<name<<endl;
        }
};
```

Object:

- Instance of a class is called object.
- Run time entity
- Has object name, data(attributes), Methods(functions)

Note:

- When a class is defined, only the specification for the object is defined;
- no memory or storage is allocated. To use the data and access functions defined in the class, we need to create objects.

Syntax: class_name object_name;
Ex:
student s;

Accessing members: . dot operator is used to access members of class

Syntax: Object-name.function-name(actual arguments);
Ex:

```
s.get_data();
s.put_data();


int main()
{
student s;
s.getdata();
s.putdata();
return 0;
}
```

Scope Resolution operator:

Scope:- Visibility or availability of a variable/ function in a program is called as scope. There are two
types of scope. i)Local scope ii)Global scope

Local scope: visibility of a variable is local to the function in which it is declared.
Global scope: visibility of a variable to all functions of a program

Scope resolution operator in " :: "

Scope resolution operator :: for variables

This is used to access global variables if same variables are declared as local and global

```
#include<iostream.h>
int a=5;                // global var

void main()
{
int a=1;                // local var
cout<<"Local a="<<a<<endl;
cout<<"Global a="<<::a<<endl;
}
```

Scope resolution operator :: for method

Member Functions in Classes
There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

```
#include <iostream>
using namespace std;
class sample
{
public:
void output();      //function declaration  but not defined
};

// function definition outside the class

void sample::output()
        {
          cout << "Function defined outside the class.\n";
        };

int main()
      {
```

```
    sample obj;
    obj.output();
    return 0;
  }
```

Output of program:
Function defined outside the class.

Feature: Abstraction and Encapsulation

Encapsulation:

- is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse.
- Data encapsulation led to the important OOP concept of data hiding.

Example:

```
#include <iostream>

using namespace std;


class EncapsulationExample

{

private:

   // we declare a as private to hide it from outside

   int a;

 public:

   // set() function to set the value of  a

   void set(int x)

      {        a = x;      }

     // get() function to return the value of a

   int get()

   {        return a;     }

};


// main function
```

```
int main()
{    EncapsulationExample e1;
     e1.set(10);
     cout<<e1.get();
   return 0;
}
```

Note: The variable a is made private so that this variable can be accessed and manipulated only by using the methods get() and set() that are present within the class. Therefore Encapuslation.

Abstraction

- Data abstraction is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

Example :

```
#include <iostream>
using namespace std;
 class Add
        {
        private:
           int a, b, c;
        public:
          void sum(int x, int y)
           {
            a = x;
            b = y;
            c = a + b;
            cout<<"Sum of the two number is : "<<c<<endl;
           }
      };
int main()
{
```

```
    Add a;
    a.sum(5, 4);
    return 0;
}
```

Note : The class 'Add' holds the private members a, b and c, which are only accessible by the member functions of that class.

Application of Abstraction and Encapsulation

Real life examples of Abstraction are

- Abstraction shows only important things to the user and hides the internal details for example when we ride a bike, we only know about how to ride bike but can not know about how it work ? and also we do not know internal functionality of bike.
- Consider a real life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car.

Real life examples of Encapsulation are

- 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. Here automatic cola vending machine is a class. It contains both data i.e. Cola can and operations i.e. service mechanism and they are wrapped/integrated under a single unit Cola Vending Machine. This is called Encapsulation.

S.NO	Abstraction	Encapsulation
1	Abstraction is the method of hiding the unwanted information.	Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
2	We can implement abstraction using abstract class and interfaces.	Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public
3	In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.

Access specifiers – public, private, protected, friend, inline

Access specifier or access modifiers are the labels that specify type of access given to members of a class. These are used for data hiding. These are also called as visibility modes. There are three types of access specifiers
1.private
2.public
3.protected

1.Private:
If the data members are declared as private access then they cannot be accessed from other functions outside the class. It can only be accessed by the functions declared within the class. It is declared by the key word **private**.

Only the member functions or the friend functions are allowed to access the private data members of a class.

2.public:
If the data members are declared public access then they can be accessed from other functions out side the class. It is declared by the key word **public**.

3.protected: The access level of protected declaration lies between public and private. This access specifier is used at the time of inheritance.

class members declared as Protected can be accessed by any subclass (derived class) of that class.

Note:-
If no access specifier is specified then it is treated by default as private.

Specifiers	Within same class	In derived Class	Outside the class
Private	Yes	No	No
Protected	Yes	Yes	No
Public	yes	Yes	Yes

Friend:

- A friend function is a function that is declared outside a class, but is capable of accessing the private and protected members of class.
- Generally, The private members cannot be accessed from outside the class. i.e a non member function cannot have an access to the private data of a class.
- In C++ a non member function can access private by making the function friendly to a class.
- Friend function is declared by using keyword “friend”

Characteristics of a Friend function:

- It can be invoked like a normal function without using the object.

- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.
- The friend function must have the class to which it is declared as friend passed to it in argument. Ex: friend float mean(Sample s), where sample is the class name in which then mean function is declared as friend
- The keyword friend is placed only in the function declaration , not in the function definition.

- **Syntax:**
  class class_name
  {
  friend data_type function_name(arguments/s);
  };

Example Program 1:

```
#include<iostream>

class employee

        {

        private:

        friend void sal();

        };

        void sal()

        {  int salary=4000;

        cout<<"Salary: "<<salary;

        }

int main()

{  employee e;

sal();

return 0;   }
```

Example 2:

// C++ program to demonstrate the working of friend function

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

class Sample
{
 private:  int a,b;
public:
void setdata()
{a=20;b=25;}

// friend function definition
friend float mean( Sample s)
{return float(s.a+s.b)/ 2.0; }
};

int main()
{
  Sample x;
   x.setdata();
  cout<< "mean value is" ;
  cout <<  mean(x) ;
    return 0;
}
```

Inline:

- Inline functions are handled just like a macro.
- When you call such a function, the function's body is directly inserted into this code.
- In other words, the inline function code is substituted at the place of the program line from which it is called
- Inline expansion makes a program run faster because the overhead of a function call and return is eliminated.

- It is defined by using key word “inline”

Example for Inline

```
#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 << "Maximum of (20,10): " << Max(20,10) << endl;
  cout << "Maximum of (0,200): " << Max(0,200) << endl;
  cout << "Maximum of (100,1010): " << Max(100,1010) << endl;
    return 0;
}
```

Output:

```
Maximum of (20,10): 20
Maximum of (0,200): 200
Maximum of (100,1010): 1010
```

Method: Constructor and Destructor

Constructors:

- Constructor is a special member function which is used to initialize the objects of its class.
- Constructor has same name as the class the class name
- Constructors don’t have return type
- A constructor is automatically called when an object is created.
- It must be placed in public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).

- A constructor is declared and defined as follows:

```
class  Wall {
```

```
  public:
    // create a constructor
    Wall() {
      // code
    }
};
```

Example Program : / C++ program to demonstrate the use of default constructor

```
#include<iostream>
using namespace std;
class sum
   { int a,b,c;
         public:
                  sum()
                  {
                  a=10;
                  b=20;
                  c=a+b;
                  cout<<"Sum: "<<c;
                  }
};

int main()
{
sum s;
return 0;}
```

Note: There is no need to write any statement to invoke the constructor function.

CHARACTERISTICS OF CONSTRUCTOR

- They should be declared in the public section.
- They are invoked automatically when the objects are created.

- They do not have return type, not even void.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other c++ functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses. They make “implicit calls‟ to the operators new and delete when memory allocation is required.

DESTRUCTORS:

- A destructor, is used to destroy the objects that have been created by a constructor.
- Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

Eg: ~item() { }

CHARACTERISTICS OF DESTRUCTOR:

- A destructor never takes any argument nor does it return any value.
- It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible.
- It is a good practice to declare destructors in a program since it releases memory space for future use.

Program for destructor

```
#include<iostream>

using namespace std;

class sum

{ int a,b,c;

public:

        sum()

        { a=10;

        b=20;

        c=a+b;

        cout<<"Sum: "<<c;        }


        ~sum()

        {  cout<<endl<<"memory released";    }

};
```

```
int main()
{
sum s;
cout<<endl<<"call main";
return 0;
}
```

Output:

```
Sum: 30
call main
memory released
```

Note: In above example when you create object of class sum auto constructor of class is call and after that control goes inside main and finally before end of program destructor is call.

OBJECT ORIENTED DESIGN CONCEPTS

Introduction

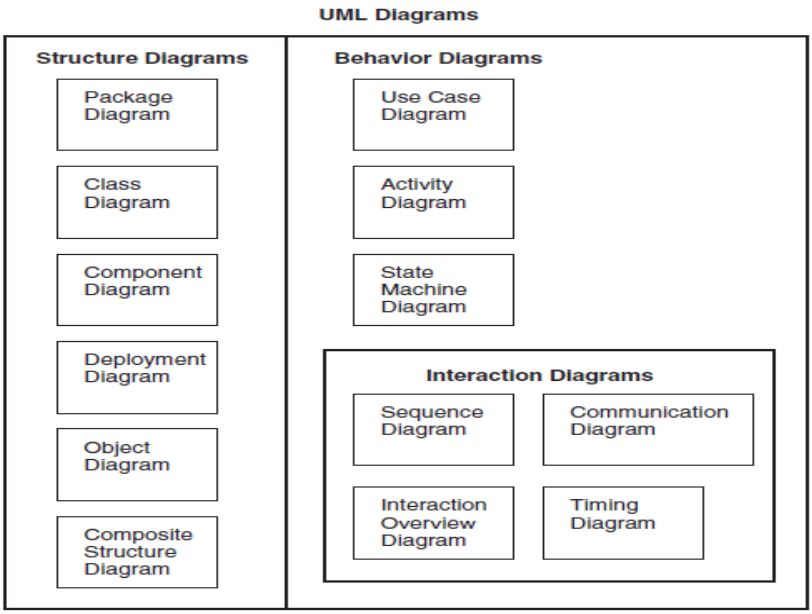
- The Unified Modeling Language (UML) is the primary modeling language used to analyze, specify, and design software systems
- In the mid-1990s, Booch, Rumbaugh, and Jacobson joined forces at Rational Software Corporation created the first version of the UML
- In November 1997 the Object Management Group (OMG), adopted the UML as a standard
- UML is not a programming language, it is rather a visual language
- It is a general-purpose modeling language, it can be utilized by all the modelers

UML Diagrams Introduction

- UML diagrams can be classified into two groups: **structure diagrams** and **behavior diagrams**
- **Structure Diagrams:**
    - These diagrams are used to show the static structure of elements in the system.
    - They may depict such things as the architectural organization of the system, the physical elements of the system, its runtime configuration, and domain-specific elements of your business, among others.

- **The UML structure diagrams include**
  - **■Package diagram**
  - **■ Class diagram**
  - **■ Component diagram**
  - **■ Deployment diagram**
  - **■ Object diagram**
  - **■ Composite structure diagram**

- **Behavior diagrams:**
  - The dynamic behavioral semantics of a problem or its implementation is depicited
  - **The UML Behavior diagrams include**
    - ■Use case diagram
    - ■ Activity diagram
    - ■ State machine diagram
    - ■ Interaction diagrams
    - ■ Sequence diagram
    - ■ Communication diagram
    - ■ Interaction overview diagram
    - ■ Timing diagram

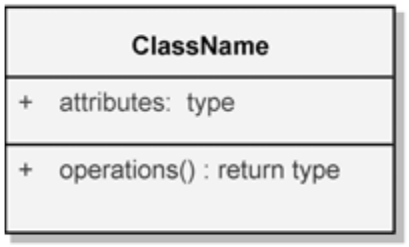


The basic building blocks of UML are:

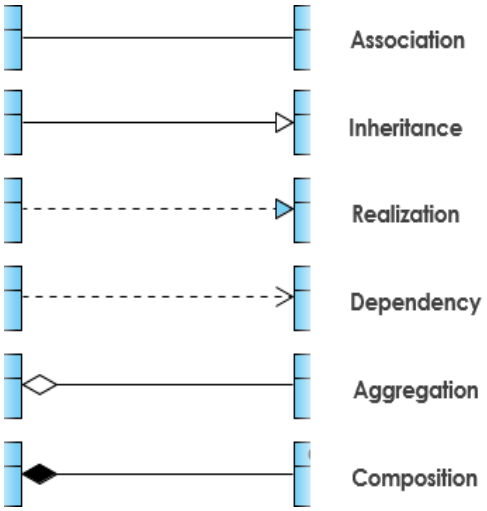
- model elements (classes, interfaces, components, use cases, etc.)
- relationships (associations, generalization, dependencies, etc.)
- diagrams (class diagrams, use case diagrams, interaction diagrams, etc.)

UML Class Diagram and its components

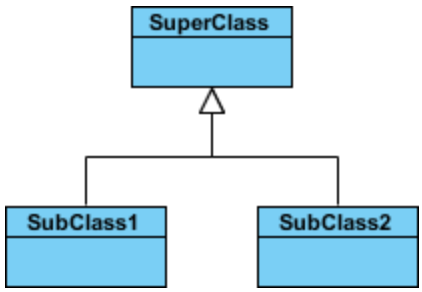
- The most widely use UML diagram is the class diagram.
- It is the building block of all object oriented software systems.
- We use class diagrams to depict the static structure of a system by showing system's classes, their methods and attributes.
- Class diagrams also help us identify relationship between different classes or objects.
- The class icon consists of three compartments 1. class name 2. attributes 3. Operations



- A name is required for each class and must be unique to its enclosing namespace.
- By convention, the name begins in capital letters, Again by convention, the first letter of the attribute and operation names is lowercase
- There are three types of modifiers that are used to decide the visibility of attributes and operations.
  ■ Public (+) Visible to any element that can see the class
  ■ Protected (#) Visible to other elements within the class and to subclasses
  ■ Private (-) Visible to other elements within the class
  Also we have other modifiers
  ■ Package (~) Visible to elements within the same package
- Classes rarely stand alone; instead, they collaborate (have relationships) with other classes in a variety of ways.
- A relationship can be one of the following types:
  - Association
  - Inheritance or Generalization
  - Realization
  - Dependency
  - Aggregation
  - Composition



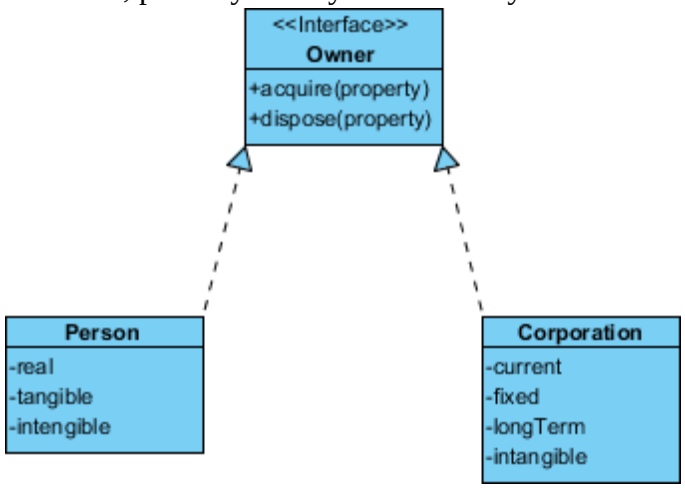
- **Association:**
  - The association icon connects two classes and denotes a semantic connection.
  - They are represented by a solid line between classes. ——————
  - Associations are typically named using a verb or verb phrase which reflects the real world problem domain.
  - A class may have an association to itself (called a reflexive association)
  - It is also possible to have more than one association between the same pair of classes
  - Associations may be further adorned with their multiplicity
  - It can form several types of associations, such as one-to-one, one-to-many, many-to-one, and many-to-many.
- **Inheritance (or) Generalization:**
  - It Represents an "is-a" relationship
  - The relationship is displayed as a solid line with a hollow arrowhead that points from the child element to the parent element..



  - The arrowhead points to the superclass, and the opposite end of the association designates the subclass.
  - The subclass inherits the structure and behavior of its superclass.

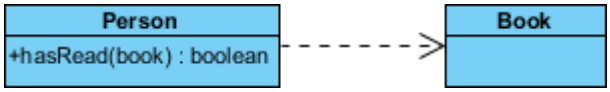
- **Aggregation:**
    - Aggregation represents a "part of" relationship.
    - The relationship is displayed as a solid line with a unfilled diamond at the association end, which is connected to the class that represents the aggregate
- **Composition:**
    - The relationship is displayed as a solid line with a filled diamond at the association end, which is connected to the class that represents the whole or composite.
    - It forms a two-way relationship.
    - It is a special case of aggregation.
    - It is known as Part-of relationship.
    - If a composite is deleted, all other parts associated with it are deleted
    - If a delete operation is executed on the folder, then it also affects all the files which are present inside the folder. All the files associated with the folder are automatically destroyed once the folder is removed from the system.

- **Realization**
    - It is a kind of relationship in which one thing specifies the behavior or a responsibility to be carried out, and the other thing carries out that behavior.
    - It can be represented on a class diagram or component diagrams.
    - It is denoted using the dashed directed line with a sizeable filled arrowhead
    - For example, the Owner interface might specify methods for acquiring property and disposing of property. The Person and Corporation classes need to implement these methods, possibly in very different ways.

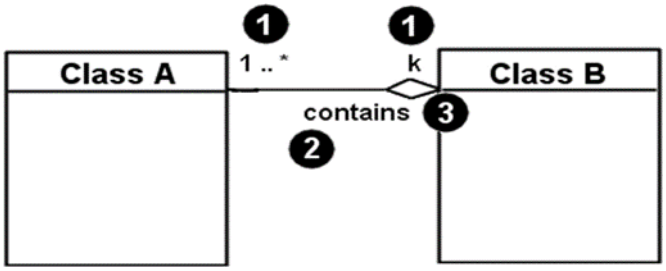


- **Dependency**

- Whenever there is a change in either the structure or the behavior of the class that affects the other class, such a relationship is termed as a dependency.
- A special type of association
- It is a unidirectional relationship.
- The relationship is displayed as a dashed line with an open arrow.



Relationships have 3 components as shown in figure



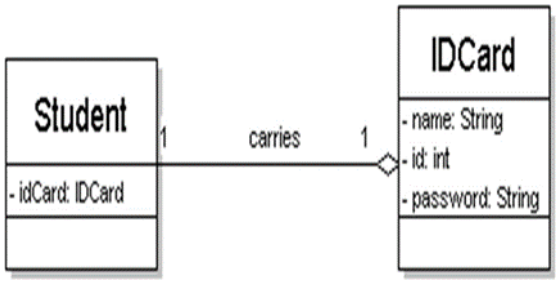
1. **Multiplicity:** an indication of how many objects may participate in the given relationship, such as one-to-one, one-to-many, many-to-one, and many-to-many.
   - **1** **Exactly one**
   - ***** **Unlimited number (zero or more)**
   - **0..*** **Zero or more**
   - **1..*** **One or more**
   - **0..1** **Zero or one**
   - **3..7** **Specified range (from three through seven, inclusive)**
   - **3..*** **3 or more (also written as "3..")**

2. Name : what relationship the objects have

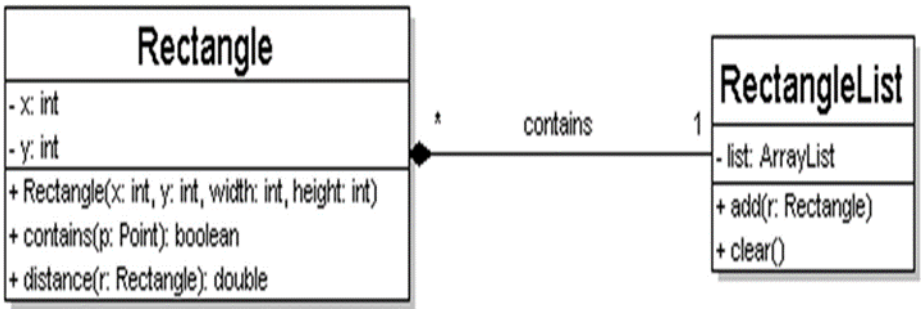
3. Navigability: direction

Multiplicity of associations

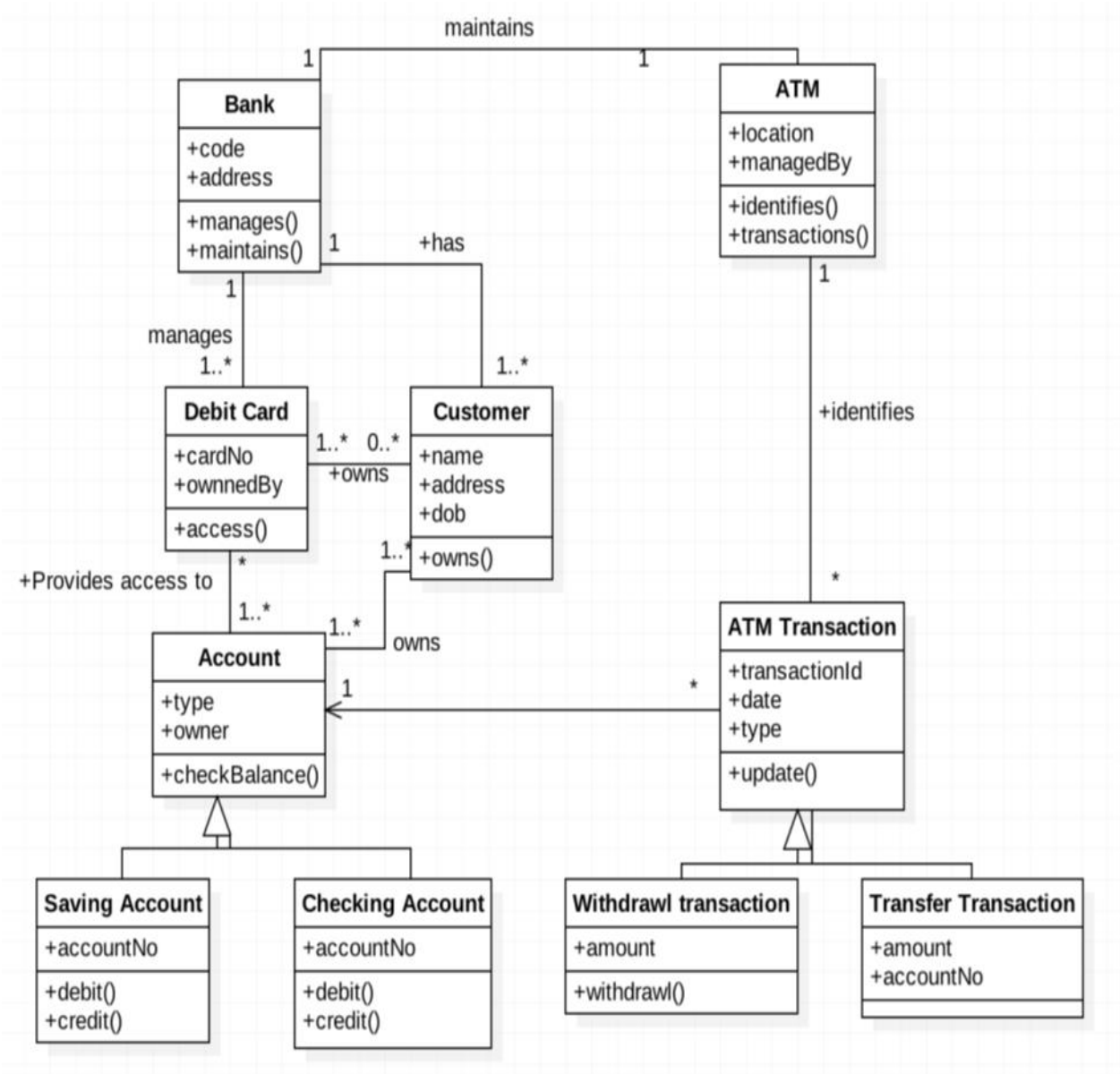
- **one-to-one : Example each student must carry exactly one ID card**



- **one-to-many :** one rectangle list can contain many rectangle



Sample of UML Class Diagrams for ATM System



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

Use Case Diagram:

- Use case diagrams describe the functionality of a system and users of the system.
- It models the dynamic behavior of the system. It is a high level diagram.
- They contain the following elements:
  1. Actors: Someone interacts with the system. actors are human users, system, organization and other internal or external applications that interacts with the subject
  2. Use cases: represents functionality or services provided by a system to users
- Use case diagram does not show the order in which steps are performed to achieve the goals of each use case.

Purpose of Use case Diagram:

- Help to identify internal and external factors that may influence the system
- Specify the context of a system
- Capture the requirements of a system
- Validate a systems architecture
- A key concept of use case modeling is that it helps us design a system from the end user's perspective.

Notations:

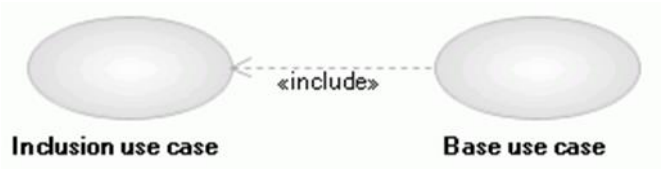
- The system boundary is the entire system and they are represented using rectangle which contains use cases.
- Actors are placed outside the system boundary and represented by actor symbol.
- Primary actors (example: customer) are represented in the left of system, secondary actors (example: Bank) placed at the right side of the system.
- Use cases the represented using oval, labeled with actions/ system functions
- Relationships: A relationship between two use cases is basically modeling the dependency between the two use cases.
- Types of relationship are

Association Relationship

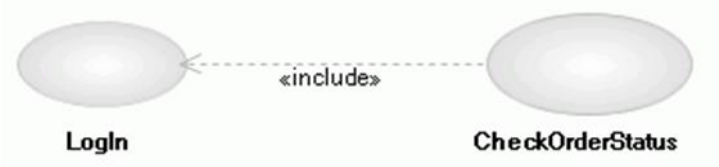
- An Association shows the participation of an Actor in a Use Case
- i.e. Relationship between the actors and use cases.
- An actor must be associated with at least one use case.
- Multiple actors can be associated with a single use case
- an association appears as a solid line between two classifiers

Include Relationship

- The include relationship adds additional functionality not specified in the base use case.
- Base use case requires an include use case to complete the function.
- An include relationship is displayed as a dashed line with an open arrow pointing from the base use case to the inclusion use case. The keyword «include» is attached to the connector.



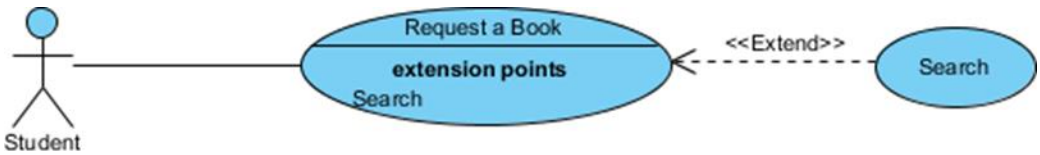
- Example Login base case requires verify password as include use case
- Checkin requires boarding pass generation.



Use Case Example - Extend Relationship

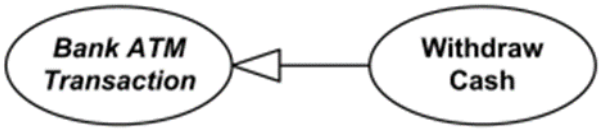
- The extend relationships represents optional functionality or system behavior.
- A subflow is executed only when criteria is met.
- An extend relationship is displayed as a dashed line with an open arrowhead pointing from the extension use case to the base use case. The arrow is labeled with the keyword «extend».

- Example: login base use case may extend Login error message as extended use case.
- Example: search extend It shows an extend connector and an extension point "Search".

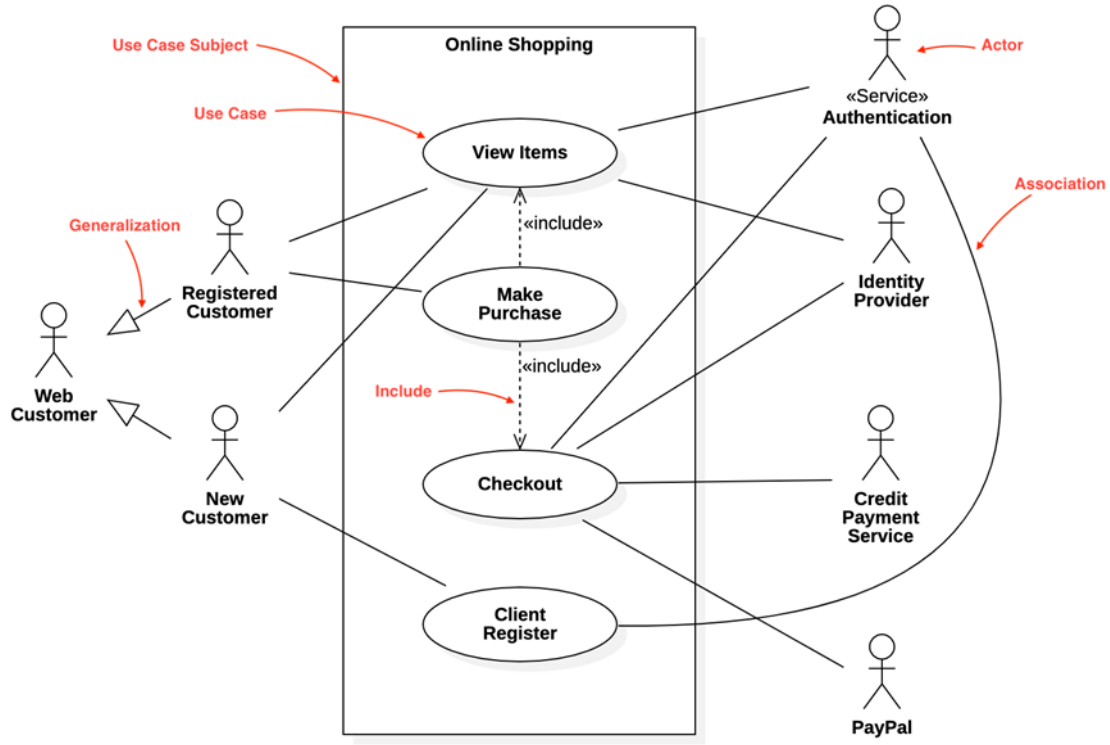


Generalization Relationship

- A generalization relationship means that a child use case inherits the behavior and meaning of the parent use case. The child may add or override the behavior of the parent.
- A generalization relationship is displayed as a solid line with a hollow arrowhead that points from the child model element to the parent model element.
- Example :



Use case Diagram for online shopping



References:

1. Reema Thareja, Object Oriented Programming with C++, 1st ed., Oxford University Press, 2015
2. https://www.programiz.com/cpp-programming
3. https://www.codesdope.com/practice/practice_cpp/
4. https://www.tutorialspoint.com/cplusplus/
5. https://www.javatpoint.com/cpp-tutorial
6. https://www.sitesbay.com/cpp/index

7. https://www.uml-diagrams.org/use-case-diagrams.html
8. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/
9. https://docs.staruml.io/working-with-uml-diagrams/use-case-diagram