Object Oriented Programming

Agenda

I. OO Vs Functional approach
II. Introduction to Object Oriented Programming
III. Object Oriented Approach
IV. C++ NON Object Orientation features
V. Foundation of Object Orientation
VI. C++ Programming: Other features

Chapter I

OO Vs Functional approach

I. OO Vs Functional approach

Agenda

1. Software Evolution
2. Bank Account Processing System
3. Functional Decomposition method
4. Object Oriented Paradigm
5. An Instance – Classes, Objects & Communication
6. An OO model instance for a Restaurant
7. OO Model Restaurant Classes

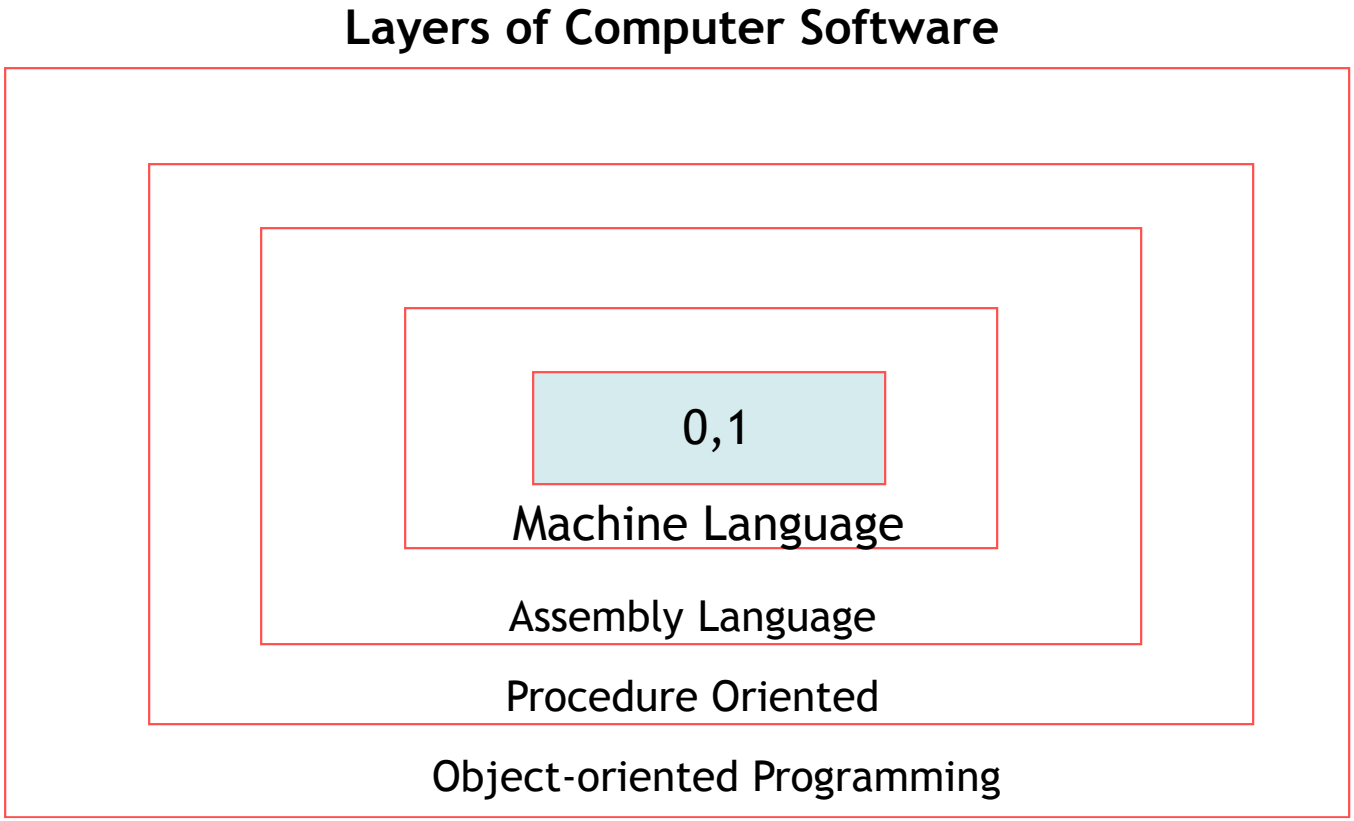
I. OO Vs Functional approach

1. Software Evolution

- Software evolution has distinct phases or layers of growth.
- Each layer represents an improvement over the previous one.
- All these layers are functional in software systems.
- Since the invention of computer systems many programming approaches have been tried such as:
  - Modular programming
  - Top-down programming
  - Bottom-up programming
  - Structured programming

I. OO Vs Functional approach

1. Software Evolution



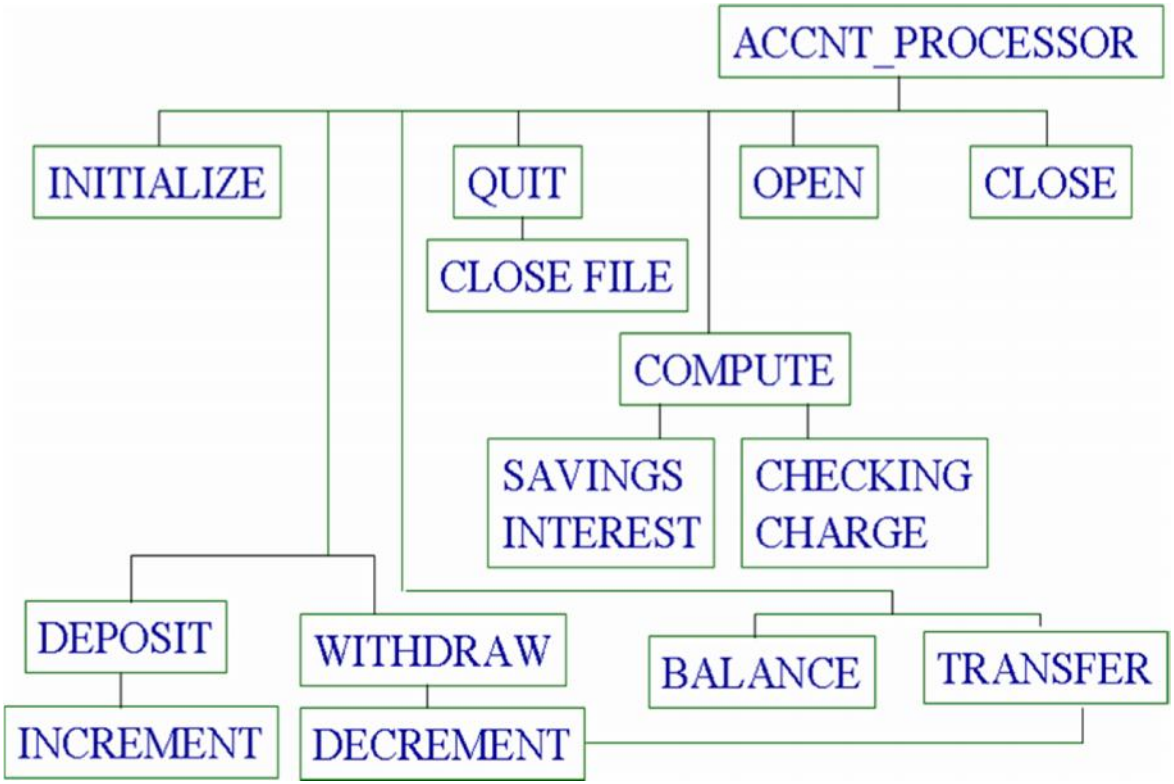
I. OO Vs Functional approach

2. Bank Account Processing System

- Account has Name, Account Number and Balance associated
- Savings and Checking (current) types of accounts
- Operations on Accounts
  - Open a new Account
  - Make a Deposit
  - Get the Balance
  - Make a Withdrawal
  - Transfer Funds
  - Close Account
- Savings Account gets Interest
- Service charges (interest) on Checking Account
- (limited) Debit balance facility for Checking accounts

I. OO Vs Functional approach

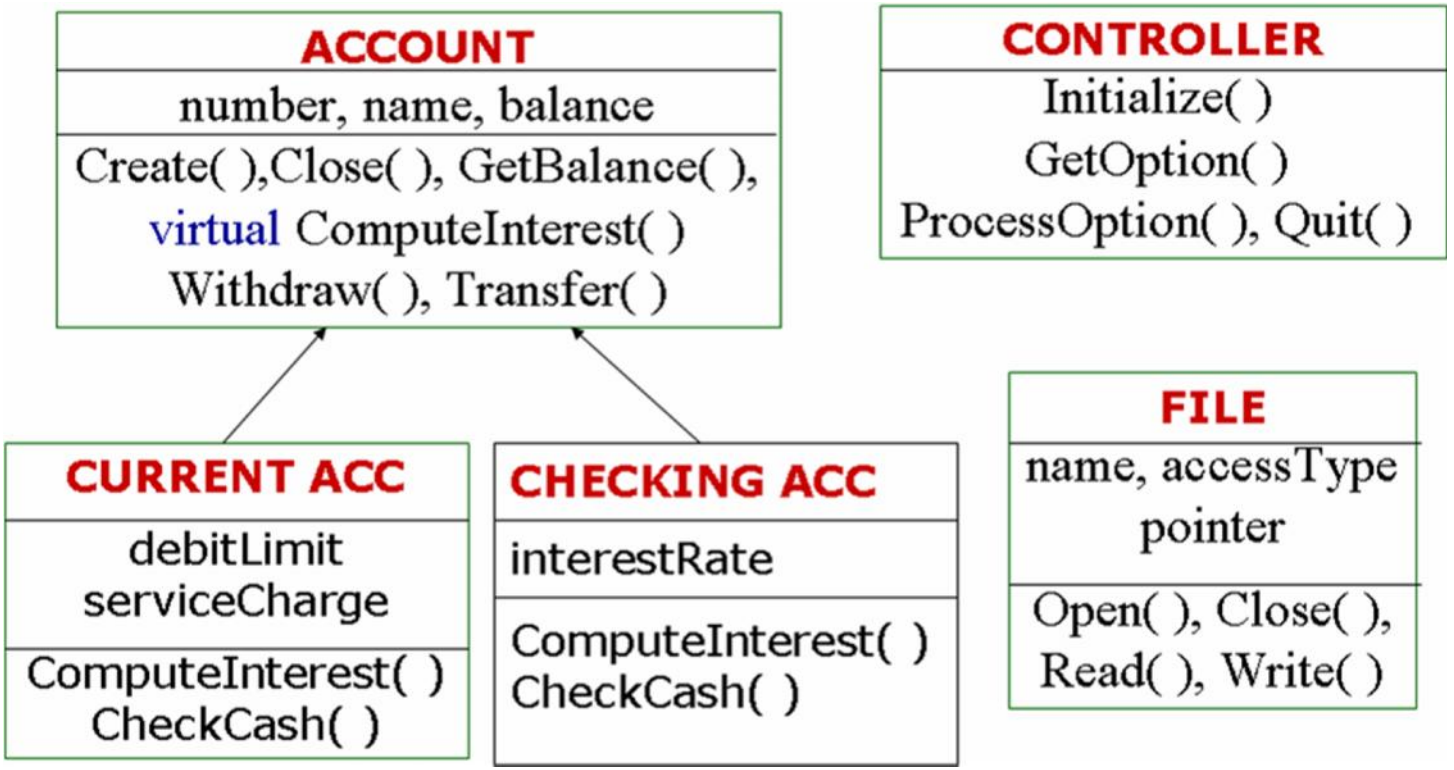
3. Functional Decomposition method



I. OO Vs Functional approach

4. Object Oriented Paradigm

organization of function & data in Object Oriented Programming



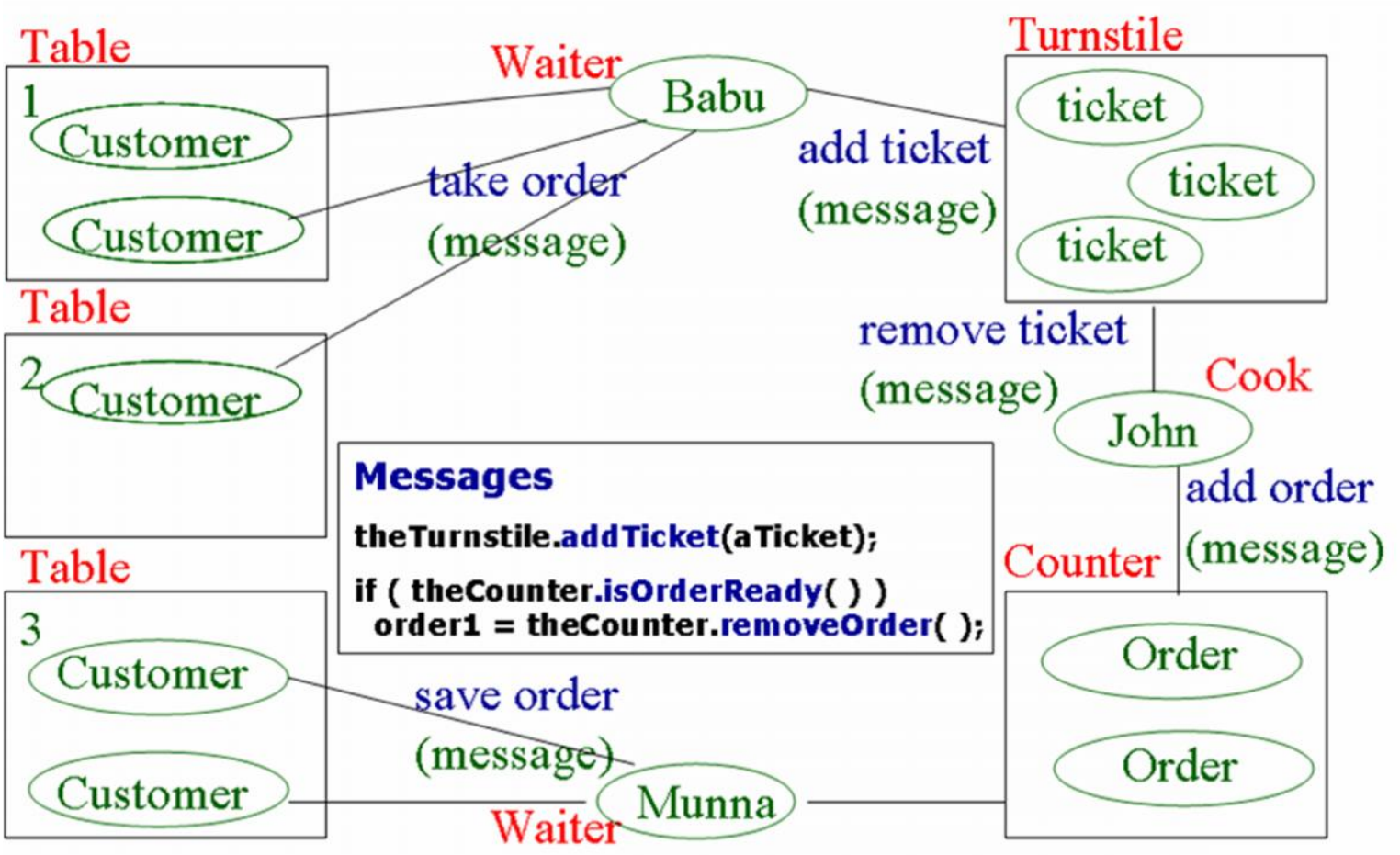
I. OO Vs Functional approach

5. An Instance – Classes, Objects & Communication

- An OO Program typically consists of many objects created of different classes.
- The objects communicate with each other by calling the member functions of each other.
- The following shows a typical object instance model in action and its corresponding class model

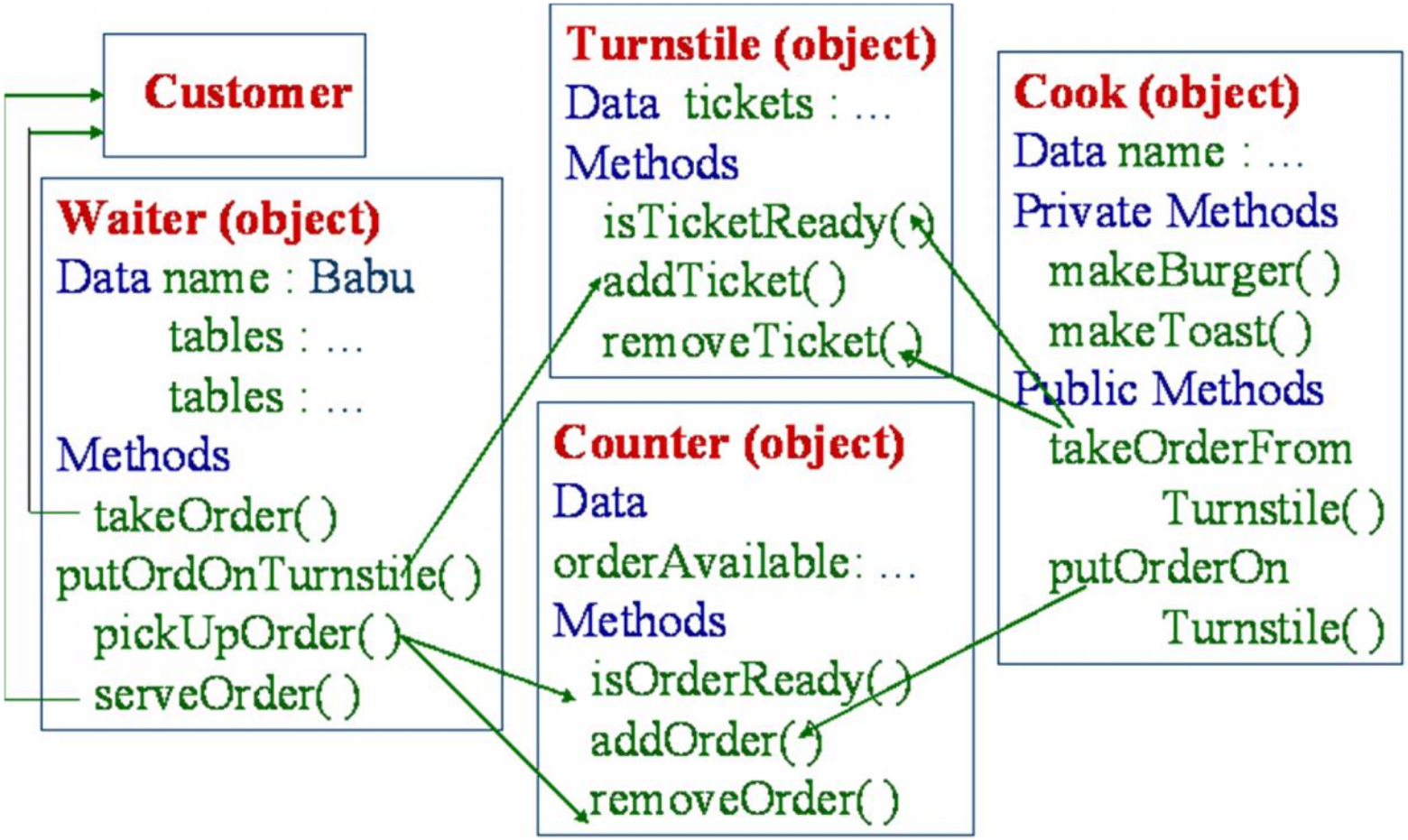
I. OO Vs Functional approach

6. An OO model instance for a Restaurant



I. OO Vs Functional approach

7. OO Model Restaurant Classes



I. OO Vs Functional approach

What we have covered:

1. Software Evolution
2. Bank Account Processing System
3. Functional Decomposition method
4. Object Oriented Approach
5. An Instance – Classes, Objects & Communication
6. An OO model instance for a Restaurant
7. OO Model Restaurant Classes

Chapter II

Introduction to Object Oriented Programming

II. Introduction to Object Oriented Programming

Agenda

1. Programming Paradigms
2. Problems with procedural programming

II. Introduction to Object Oriented Programming

1. Programming Paradigms

- The old belief

  "Whatever richness is there, it has to be in the thought itself, the language is merely a tool, a medium or a vehicle for expression of thought!"

  Was challenged and changed by an agreement

  "Quiet often, languages influence even our thinking habits!"

  As far as programming languages go, this means:

  If we need to implement the system in C++ or any OO language, then the thinking of problem formulation or modeling also should be done the OO way…

II. Introduction to Object Oriented Programming

2. Problems with Procedural Programming

- 'Code First, Data Later' approach
- Data Validation becomes a difficult job
- Reuse of code not possible in a systematic manner
- Maintenance, changes requires a lot of efforts
- There is a lot of 'Semantic Gap' between the models of reality and its representation in computer system, resulting in systems difficult to understand

II. Introduction to Object Oriented Programming

What we have covered:

1. Programming Paradigms
2. Problems with procedural programming

Chapter III

Object Oriented Approach

III. Object Oriented Approach

Agenda

1. OO Construction Approach
2. OO Approach Benefits

III. Object Oriented Approach

1. OO Construction Approach

After identifying Objects in the System

Build Software As
'Layers (hierarchy)' & 'Assembly' of (Objects)

Rather than

Developing from scratch
Using 'raw material' (Individual independent functions)

Program behavior is seen as Objects sending messages to other Objects (call functions) to get any job done.

III. Object Oriented Approach

2. OO Approach Benefits

- Data and Code 'Equal and Together' approach
- Both are important (unlike procedural approach)
- Promotes re-use, hence protecting an investment and higher productivity
  - e.g. Once 'Employee' class is developed, then the same can be (re) used in Payroll, H-R system, Project Management System etc
- Using pre-developed and pre-tested Classes means more reliable system.

III. Object Oriented Approach

2. OO Approach Benefits

- Changes happen only locally, hence avoiding complexity of maintenance
  - e.g. If the rules of 'computation of salary' are changed, then only the code of class 'Employee' will undergo changes.
- Real life entities almost map individually as Objects into a computer system.
  - e.g. Model of a 'Purchase Order System' in reality will result into 'Supplier', 'Item', 'Order' Objects.
- Due to this correspondence between objects in reality and computer representation, the semantic gap between the reality and its model reduces, hence resulting into systems easier to understand.

III. Object Oriented Approach

What we have covered:

1. OO Construction Approach
2. OO Approach Benefits

Chapter IV

C++ Non Object Orientation Features

IV. C++ Non Object Orientation Features

Agenda

1. Constants
2. Functions with Default parameters
3. cin and cout for I/O
4. References
5. Inline Functions

IV. C++ Non Object Orientation Features

1. Constants

Specifies that a variable's value is constant

Tells compiler to prevent the programmer from modifying it

Example:

```
const int i = 5;
i = 10;     // Error
i ++;       // Error
```

IV. C++ Non Object Orientation Features

1. Constants

- Const data: const int i = 5;
  - Size of an array can be specified with a const variable
    const int SIZE = 1000;
  - char Arr [SIZE]; // Allowed in C++ / Not in C
  - const keyword in function prototype
    SomeFunction (int & id) // id can get changed accidentally

    SomeFunction (const int & id); // id can NOT get changed

  - Function returning a const data
    const int f (……..)

IV. C++ Non Object Orientation Features

1. Constants

```
const int num = 10;   // declaration time initialization
num = 15; // ERROR, num is a constant

const char * ptrc = "Hi";        // pointer to constant character
ptrc[0] = 'h';                   // ERROR, pointer is pointing to constant
data
ptrc = "Hello";       // OK, we can make it point to something else

char * const cptr = "Hi";        // constant pointer to character
cptr[0] = 'h';                   // OK, pointer NOT is pointing to constant
data
cptr = "Hello";       // ERROR, pointer (contents of variable cpt) is
constant

const char * const cptrc = "Rigid"; // constant pointer to character
constant
cptrc[0] = 'r';                     // ERROR
cptrc = "Flexible";  // ERROR
```

IV. C++ Non Object Orientation Features

2. Functions with default parameters

```
// function with last argument having a default value
void ShowMessage( int atRow, int atCol, char msg[ ] = "Press ENTER to continue" )
{
}
ShowMessage( 5, 10 );      // the default message will be shown
ShowMessage( 10, 15, "Employee Record Updated" );   // this message will be shown
```

- Default values are taken only where parameters are not supplied.
- Any number of parameters may have default values, however, it should be continuous list of defaults from rightmost parameter to the leftmost. e.g. we can't have only atRow and msg having default values.

IV. C++ Non Object Orientation Features

3. cin and cout for I/O

```
int num;

// accept a number form keyboard
cin >> num;
// display the output on screen
cout << num;
```

IV. C++ Non Object Orientation Features

4. References

```
int num = 10;
int & p = num;              // p is a reference to an integer type of variable

// now p and num (two different names) refer to the SAME MEMORY location
p = 20;
cout<< “Num: <<num<<endl<< num ;              // output will be 20
```

Parameter passing using reference **(Call by Reference)**

```
void main( )
{
  int num = 10;

  F( num );
  cout<< “Num: ”<< num<<endl ;   // output will be 20
}

void F( int & p )           // upon call, p and num will be referring to the same memory loc
{
  p = 20;       // using references, cryptic code of using pointer is avoided !
}
```

IV. C++ Non Object Orientation Features

5. Inline Functions

Code for inline functions gets substituted at the place of the function call.

```
inline int FindMax( int a, int b )
{
  return (a > b ? a : b);
}

void main( )
{
    int result;
    result = FindMax( 50, 100 );
}
```

Member functions of class can also be defined as inline.

IV. C++ Non Object Orientation Features

5. Inline Functions

- Inline functions improve performance, since the function call overhead is removed.
- It works like a 'C' language macro but does not have the problems associated with macros.
  - Macros may require parenthesis to avoid ambiguity
  - Macros do not do type checking
- Inline keyword is only a compiler directive, compiler may decide against inline substitution, if it does not see any performance improvement.

IV. C++ Non Object Orientation Features

What we have covered:

1. Constants
2. Functions with Default parameters
3. cin and cout for I/O
4. References
5. Inline Functions

Chapter V

Foundation of Object Orientation

V. Foundation of Object Orientation

Agenda

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Composition
5. Association

V. Foundation of Object Orientation

1. Encapsulation

Abstract entity Class

that encapsulates

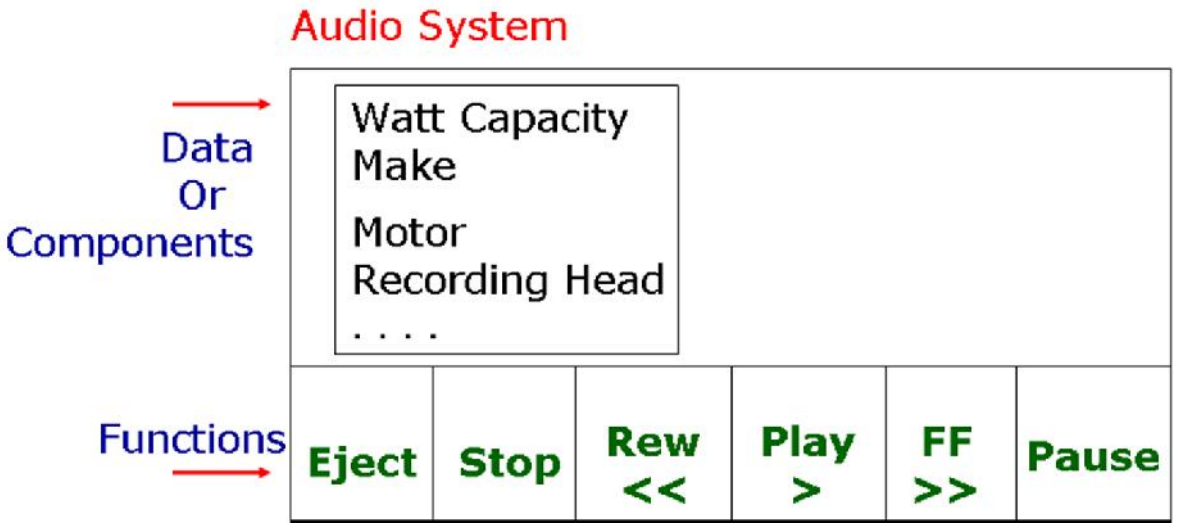
DATA and FUNCTIONS that operates on data.

These data elements & functions are called as members of class.

Data (values of variables) represents state of an object.

Data is protected by the functions of that class and access to the data is always routed via these functions.

V. Foundation of Object Orientation



1. Encapsulation

- Real life Example
  - We operate the system using the public interfaces (buttons) exposed to us.
  - The parts and the implementation of operations is hidden inside the box
  - We press buttons (stimulus to object) & get desired functionality (response).

V. Foundation of Object Orientation

1. Encapsulation

- Public, Private and Protected variables
  - The public keyword in the declaration statement specifies that the elements are accessible from code anywhere
  - The private keyword in the declaration statement specifies that the elements are accessible only from within the same class
  - The protected keyword in the declaration statement specify that the elements are accessible either from derived classes or from within the same class

V. Foundation of Object Orientation

1. Encapsulation

- C++ Implementation

```
class Employee
{
        private:
                int number;
                char name[50];
                float basicSalary;
        public:
                void SetName( char nm[ ] ) // a mutator function; used to SET data
                {
                  strcpy( name, nm );
                }
                char * GetName() // an accessor function; used to GET (return) data
                {
                  return name;
                }
                //.. other Accessor and Mutator methods for members - number and basicSalary
                int ComputeSalary( )     // compute gross salary using rules and return
                {
                }
};
```

V. Foundation of Object Orientation

1. Encapsulation

- C++ Implementation

```
#include <iostream.h>

void main( )
{
        Employee e1, e2;          // creating objects of class Employee
                                  // objects e1 and e2 will have their own data members

        // individual members of an object are referred to using a DOT notation
        // i.e. objectName.memberName

        strcpy( e1.name, "John" ); // compilation ERROR; name is a private member

        // use mutator functions to SET data inside an object
        e1.SetName( "John" );                    // OK

        // use accessor functions to GET data present inside an object
        cout<< "Name is :"<< e1.GetName( ) ;  // OK
}
```

V. Foundation of Object Orientation

1. Encapsulation

- C++ Implementation
  - Concept of this pointer:
    - Present in every non-static member function of class
    - Data type is Class *
    - Constant pointer : Value cant be changed
    - Initialized to the base address of invoking object and passed implicitly by the compiler to every non-static member function
    - Is a key-word

V. Foundation of Object Orientation

1. Encapsulation

- C++ Implementation

This Pointer

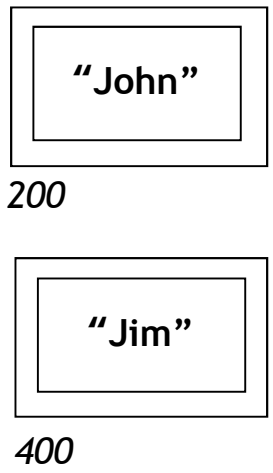
this = 200

```
void GetName( char nm[ ] )
{
  strcpy( nm, this->name );
}
```

this = 400

```
void GetName( char nm[ ] )
{
  strcpy( nm, this->name );
}
```

```
void main()
{
    Employee e1,e2;
    //set name of first employee
    e1.SetName("John");
    //set name of first employee
    e2.SetName("Jim");
    //Display names of both employee
    //GetName works for e1
    cout<<"Name is"<<e1.GetName();
    //GetName works for e2
    cout<<"Name is"<<e2.GetName();
}
```



V. Foundation of Object Orientation

1. Encapsulation

- C++ Implementation

  Any Class will have a
  Private section defining normally only the data members
  Public section defining only the member functions.

  Member Functions will be of type

Accessor	(for every data member)
Mutator	(for every data member)
Operations on data	(like ComputeSalary)
Constructor	(will be learning ahead)
Destructor	(will be learning ahead)

V. Foundation of Object Orientation

1. Encapsulation

- Benefits
  - Since generally only one function (mutator) is responsible for setting a data element, the validations are easier to introduce.
    - e.g. Validation such as 'employee number can't be negative' will be introduced ONLY inside the SetNumber( ) function of Employee class. This function will be the ONLY function in entire system responsible for setting employee number.
  - Implementations of functions can be changed without affecting the program code that uses it. So, changes happen locally and can be introduced easily even after the system is developed and operational.
    - e.g. If the rules of computations of salary are changed at any point, ONLY the function ComputeSalary( ) will undergo changes.
  - All the programs that just make a call to function ComputeSalary( ) will only be required to be compiled (linked with new version of class Employee) and will not undergo changes.

V. Foundation of Object Orientation

1. Encapsulation

- C++ Class : Other features

Class – Defining functions outside

Functions can be declared inside but defined outside.

```
class Employee
{
      private:
       // data members
      public:
       void SetNumber( int n );
       // other functions
};

void Employee::SetNumber( int n )
{
  number = n;
}
```

In fact, the professional programming practice is that ALL the functions are declared inside the class but defined outside.

V. Foundation of Object Orientation

1. Encapsulation

- C++ Class : Other features

Class – Constructor member function

Constructor method has the same name as that of the class and is automatically called, when an object of a class is instantiated. It does not return any value.

```
class Employee
{
        private:
                int number;
                char name[50];
                // ….. other data members
        public:
                Employee( )
                {
                  strcpy( name, “unknown” );
                }
                // other functions
};
void main( )
{
        Employee e1;
        // name of e1 will have string “unknown”
}
```

V. Foundation of Object Orientation

1. Encapsulation

- C++ Class : Other features
  - Constructor can be overloaded.
  - It is not required to supply parameters while creating objects, **ONLY IF** constructor is NOT defined **OR** there is zero argument constructor present **OR** constructor has parameters but all default values

```
class Employee
{
        public:
                Employee( )
                {
                  strcpy( name, “unknown” );
                }
                Employee( int no, char nm[ ] )
                {
                  number = no;
                  strcpy( name, nm );
                }
};
```

V. Foundation of Object Orientation

1. Encapsulation

- C++ Class : Other features

Class – Destructor member function

Destructor method has the same name (but preceding with tilde character) as that of the class and is automatically called, when an object of a class is deleted or goes out of scope. Only dynamically created objects can be deleted by programmer.

```
class Employee
{
        private:  // ….. data members
        public:  // … member functions
                ~Employee( )
                {
                  // perform any application specific action, before the object is deleted
                }
};
void main( )  // objects created and deleted dynamically using new and delete
{
        Employee * ep;
        ep = new Employee( );  // constructor called and ep will point to object created
        // …..use this object using ep, such as ep->SetName( )… ep->ComputeSalary( ) etc
        delete ep;                          // destructor called since object getting deleted
}
```

V. Foundation of Object Orientation

1. Encapsulation

- C++ Class : Static Data and Functions
  - Data to be shared by all objects is stored in static data members
  - Is a class variable
  - Only one copy created, shared by all objects
  - Can be accessed without object, using class name

```
class Complex
{
    int Real;
    int Imag;
public:

    static int Count;

    int GetReal()const;// const
    void SetReal(int); // non-const
};
void main (void)
{
    Complex::Count =10;
}
```

```
#include"complex.h"

int Complex::Count;

int Complex::GetReal()const
{
    return Real;
}
```

V. Foundation of Object Orientation

2. Inheritance

Hierarchy of Classes where CHILD class inherits data & function members of PARENT class with an ability to add its own data and function members

---

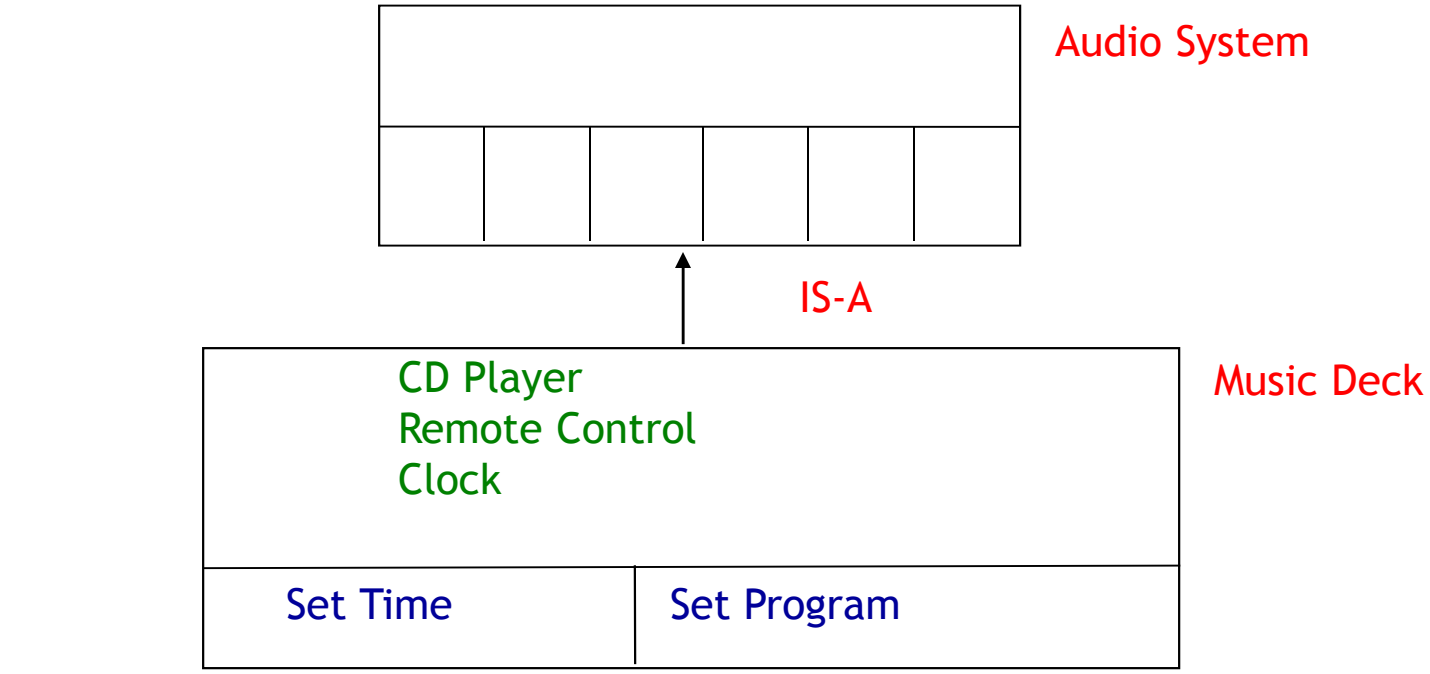
Terminology

Inherited class	Inheriting class
Parent	Child
Super	Sub
Base	Derived (we will be using this)

V. Foundation of Object Orientation

2. Inheritance

- Real life example

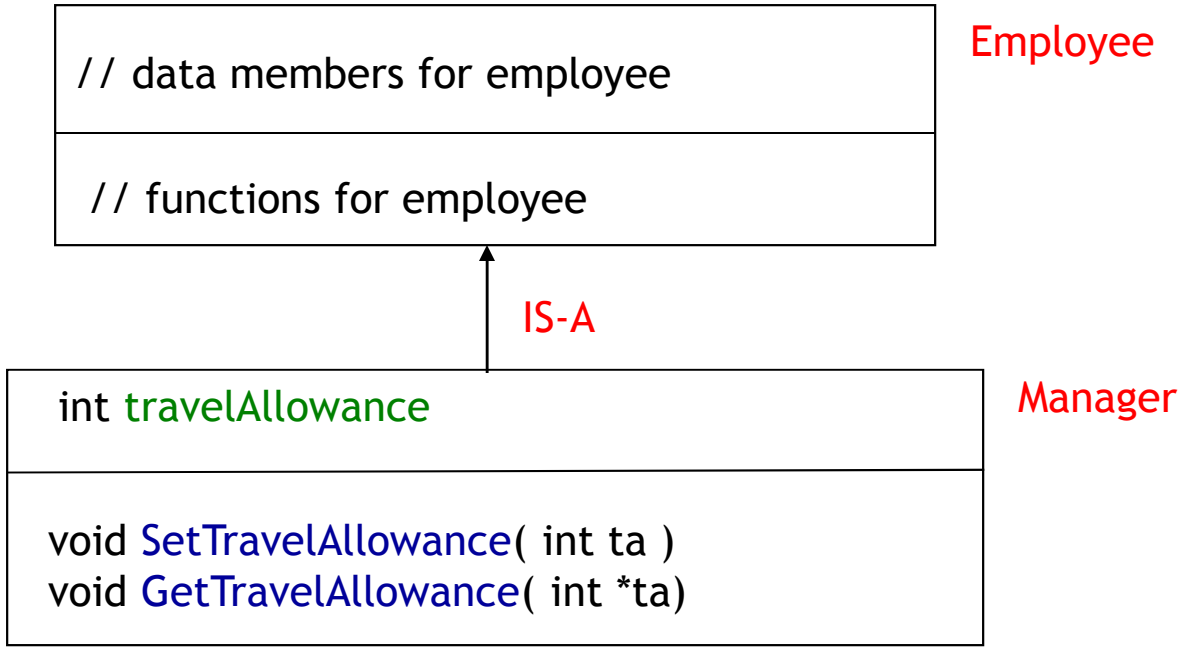


Music Deck IS-An Audio System
with more features like CD Player, Clock….
and more functions like Setting Time, Setting-Program

V. Foundation of Object Orientation

2. Inheritance

- C++ Implementation



Manager IS-An Employee, so it inherits all the characteristics (data members and functions) of an Employee and extends it further with a data member travelAllowance and it corresponding functions.

V. Foundation of Object Orientation

2. Inheritance

- Other features

Class – Protected Section

```
class A
{
    private:
        // members
    protected:
   int x;
    public:
        // members
};
```

The protected section is like private section, as far as accessibility from the outside is concerned. i.e. the protected members can not be accessed from outside.

However, they are accessible from the derived class.
Private members of the base class are not accessible even from the derived class.

V. Foundation of Object Orientation

2. Inheritance

- Public / Private /Protected

Inheritance (Access Modifiers) can either be public, private or protected.
It has to be specified while declaring the derived class.
e.g. class Manager : public Employee

```
{
        // ….
  };
```

This mode of inheritance decides the accessibility status of the inherited members of the base class in the derived class.

Following summarizes the same:
Public Inheritance: Public/Private/Protected members of Employee will go in corresponding sections of Manager class. So, base class members retain their status in inheritance.

Private Inheritance: All the inherited members of the Employee Class will go into the private section of Manager class.

Protected Inheritance: All the inherited members of the Employee Class will go into the protected section of Manager class.

V. Foundation of Object Orientation

2. Inheritance

- C++ Implementation

```
class Employee              // same as what we defined earlier
{
};

class Manager : public Employee
{
        private:
                int travelAllowance;
        public:
                void SetTravelAllowance( int ta )
                {
                  travelAllowance = ta;
                }
                void GetTravelAllowance( int * ta )
                {
                  * ta = travelAllowance;
                }
};
```

The keyword public (while inheriting) means, members of Employee retain their corresponding accessibility status (public and private), when they are inherited by Manager class. i.e. private members of Employee will go into private section of Manager and public members of employee will go into public section of Manager.

V. Foundation of Object Orientation

2. Inheritance

- Inheritance Benefits
  - Facilitates systematic 'Re-use'.

    Code / Design can be reused within / across applications.

    This helps in increasing the productivity.

  - Inheritance forms a basis for Polymorphism (explained ahead). We are able to treat objects similarly and differently as and when required.

  - New characteristics and functionality can be easily introduced by inheriting suitable existing classes, introducing data members and extending OR over-riding the member functions.

V. Foundation of Object Orientation

3. Polymorphism

- C++ Implementation

  Compile time Polymorphism in C++

  Multiple functions having a same name but they differ in terms of a number or types of parameters. This is also called as 'Function Overloading'.

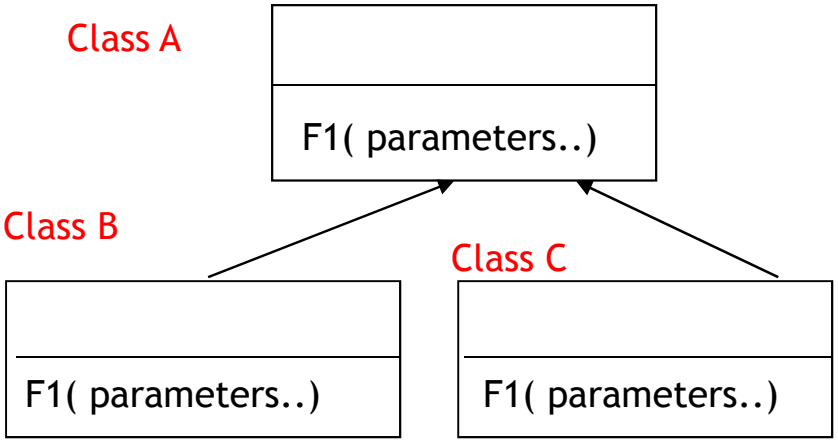
```
void Greet( )
{
  cout<< "Hello ! How are you ?"<<endl ;
}

void Greet( char message[ ] )
{
        cout<<endl<<message;
}

void main( )
{
        Greet( "Good Morning !" );   // Greet function with char [ ] parameter is called
        …
        Greet( );                    // Greet function with zero parameters is called
}
```

This polymorphic reference is resolved at compile time and compiler makes an arrangement for appropriate function call during the compilation process.

V. Foundation of Object Orientation

3. Polymorphism

- C++ Implementation
  - Compile time Polymorphism in C++ using Function Overriding



Function of a base class is also present in the derived class.

This function has the same prototype as that of the base class function but different implementation.

Derived class can call the base class method using class name as a qualifier.
e.g. from class B, it can refer to F1 of class A like:
A::F1(….)

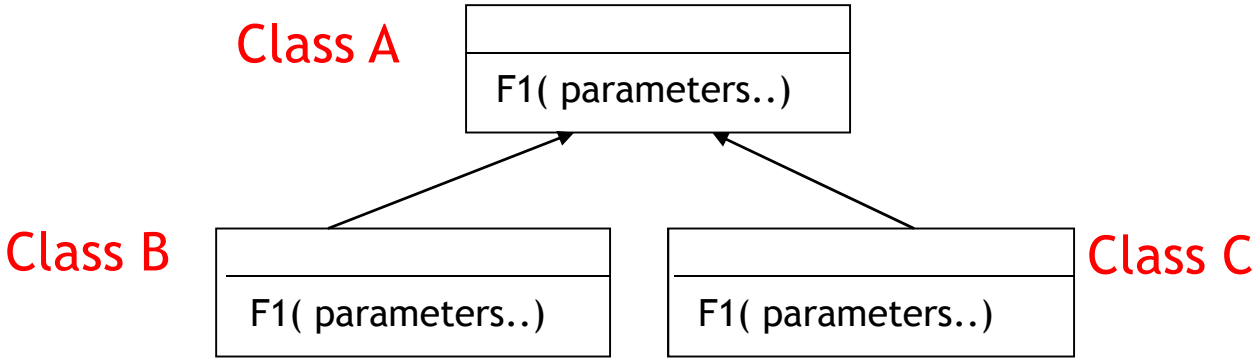
This is called as 'Function Overriding'. If function F1( ) is called on an object of the class B, then F1( ) of class B will get called.

```
A a1;
B b1;
a1.F1( );       // F1( ) of class A is called
b1.F1( );       // F1( ) of class B is called, this is resolved at compile time
```

V. Foundation of Object Orientation

3. Polymorphism

- C++ Implementation
  - Run time Polymorphism in C++ using Function Overriding and Virtual Functions

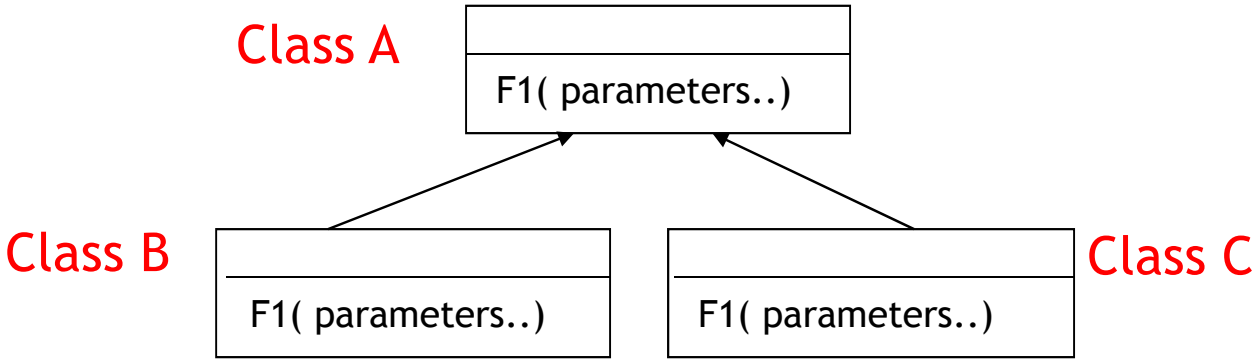



```
A * ptrA;   B b1;   C c1;

ptrA = & b1;   // a valid assignment, since class B    IS_A    class A

ptrA->F1( );       // function F1( ) of B is called, since AT THIS point of time,
                        // this pointer is pointing to an object of class B
```

V. Foundation of Object Orientation

3. Polymorphism

- C++ Implementation
  - Run time Polymorphism in C++ using Function Overriding and Virtual Functions



This reference is resolved and decision to call a specific function (of a particular class) is made during run time, hence called as **'Run time polymorphism'**.

This is done using dynamic binding and to use this feature, **function F1( ) in base class** (class A) has to be declared with a keyword **virtual**.

V. Foundation of Object Orientation

3. Polymorphism

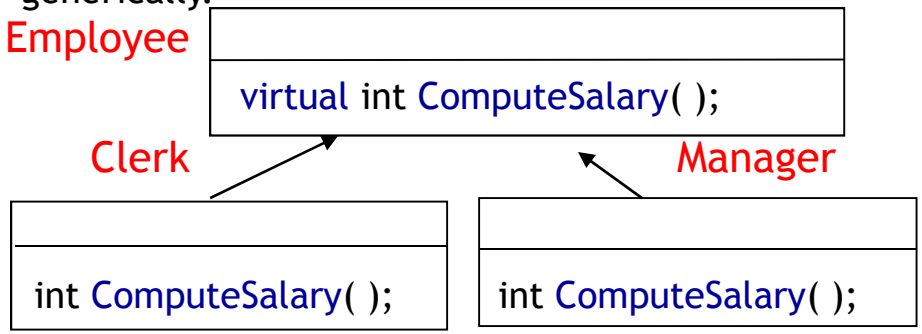
- C++ Implementation
  - Pure Virtual Function & Abstract Class
  - Pure virtual function is a virtual function that has a prototype defined but NOT the implementation or body (code) of the function.
    - It is defined using the following manner (function prototype=0).
    - void F(int x, int y)**=0**;
    - Any class that has even one pure virtual function, becomes an Abstract class and objects of Abstract class can't be created. A pointer to abstract class, however, can be declared and one can make it point to an object of a concrete (non-abstract) class.
    - The derived class has to implement ALL the pure virtual functions (provide a concrete implementation), else, it would also become an abstract class.
    - These pure virtual functions enforce a discipline on the prototype of the function. All the derived classes will have to write such functions and also with the same name and the parameters.

V. Foundation of Object Orientation

3. Polymorphism

- C++ Implementation

Processing objects in inheritance hierarchy generically.



Possible implementation of ComputeSalary of Employee class

```
virtual int ComputeSalary( )
{
    int grossSal, DA;

    DA =  * grossSal;
    grossSal = basSal +DA;

    return grossSal;
 }
```

Possible implementation of ComputeSalary of Manager class

```
int ComputeSalary( )
{
    int grossSal, empSal;
    // use Base Class method first
    empSal = Emp::ComputeSalary();
    grossSal = regularSal +  travelAllowance;  // manager specific processing
    return grossSal;
 }
```

V. Foundation of Object Orientation

3. Polymorphism

- C++ Implementation

Similarly, other classes may define their own process of computation of salary for that type of employee.

Now, application program can treat all objects having their base class similarly, as and when required.

```
Employee * ep[10];

ep[0] = new Manager( );              // a perfectly valid assignment, as explained earlier
ep[0]->SetNumber( ); // similarly set other data members

ep[1] = new Clerk( );
ep[1]->SetNumber( ); // similarly set other data members
// … similarly, create different types of employee objects

// Power of polymorphism – programmer treating all employees similarly for processing !
for ( int i = 0; I <= 9; ++i )
{
  cout<< “Name”<< ep[i]->GetName( )<<endl<<“Salary “<<
   ep[i]->ComputeSalary( );
}
```

V. Foundation of Object Orientation

3. Polymorphism

- Benefits
  - By treating the objects belonging to the same inheritance hierarchy similarly, the resultant code is elegant or easier to understand. It also results into reduction of lines of code and avoiding multiple switch / case statements on different types.

  - Polymorphism allows client programs to be written based only on the abstract interfaces (pure virtual functions in C++) of the objects which will be manipulated (interface inheritance).

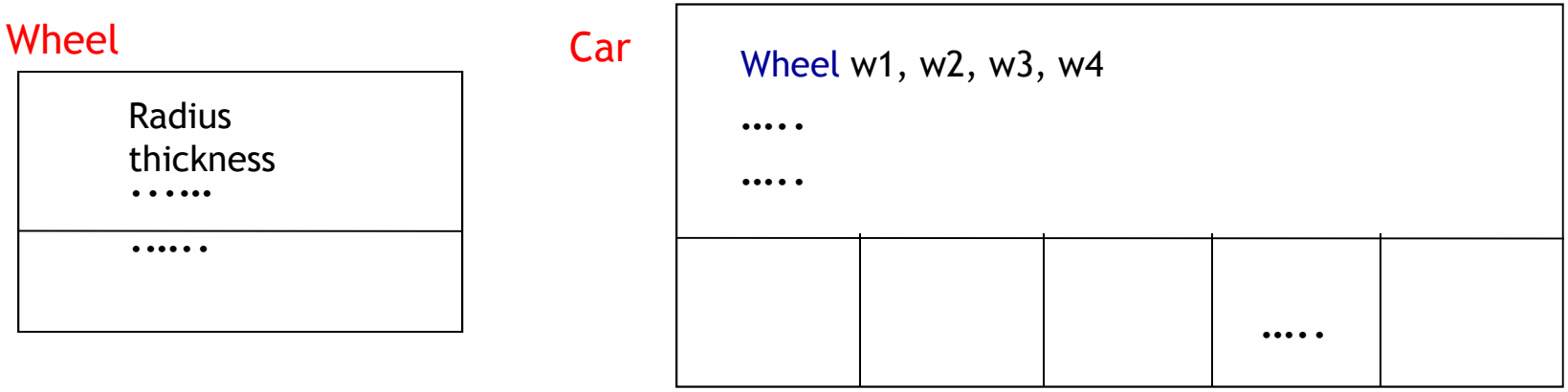
    This means that future extension in the form of new types of objects is easy, if the new objects conform to the original interface. For such additions or deletion of classes (in same hierarchy), application code is not affected in a major way.

However, there is some cost associated. Virtual functions based runtime polymorphism slows down the performance a little bit, since it involves automatic type checking at runtime for function resolution.
So, use virtual functions for specific requirement.

V. Foundation of Object Orientation

4. Composition

- Also called as Containment / Aggregation.
- This is indicated when an object contains another object.
- Represented by HAS_A relationship.



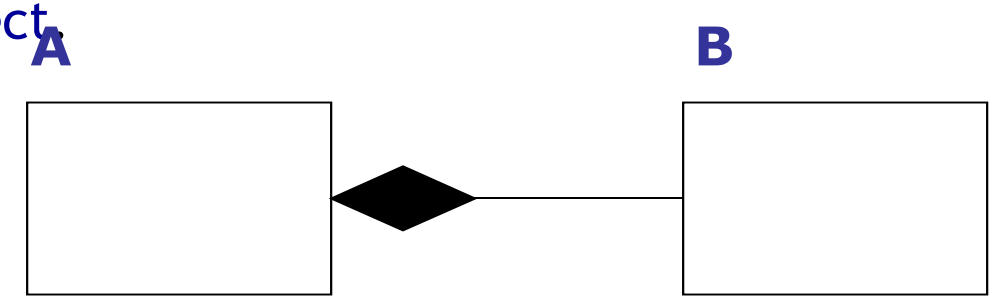
Car contains 4 objects of class wheel as data members.
These are also called as member objects.

V. Foundation of Object Orientation

4. Composition

- The above is interpreted as class A has an object of class B
- Member objects can be placed either in public or in private section of the container class.
  - e. g. A a1; a declaration inside the class B.
- If a member object is placed inside a private section of the container class, then the public methods of the member object will also be NOT available outside the class.

In this case, it will be the responsibility of the container class to provide the required functions to give an access to operate on the member object.



V. Foundation of Object Orientation

5. Association

- This is used when classes have a conceptual relationship and a loose coupling between the objects in that relationship.

- Application requirement also expects to de-link an object from a relationship instance and link it with some other object in a relationship.

- Association in C++ implemented using a pointer / reference to the object of the other class in that association relationship.

  e.g. Department * dp; declaration inside Employee class
  (only one department associated with any employee)
  and
  Employee * ep[100]; type declaration inside Department class.
  (many employees associated with a department)

V. Foundation of Object Orientation

What we have covered

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Composition
5. Association

Chapter VI

- C++ Programming: Other features

VI. C++ Programming: Other features

Agenda

1. Nested Class
2. Other OO Modeling Concepts
3. Inheritance – Aggregation – Association

VI. C++ Programming: Other features

1. Nested Class

File ”InnerclassDemo.h”

```
class Outer
{
 public:
            void ShowOut();
 private:
            int out;
            class Inner
            {
               public:
                        void ShowIn( );
               private:
                        int in;
                        static int innerstatic;
            };
};
```

VI. C++ Programming: Other features

1. Nested Class

```
#include “InnerclassDemo.h”
#include <iostream.h>            // for using cin, cout objects

void Outer::ShowOut( )
{
 Outer::out = 10;
 cout << “Outer class” << endl;
 cout << “out: “ << out << endl;

 // Inner iObj;
 // iObj.ShowIn( );
;
File “InnerclassDemoDef.cpp”
void Outer::Inner::  ShowIn( )
{
 cout << “static : “ << innerstatic << endl;
 cout << “Inner class “ << endl;
}

int Outer::Inner::innerstatic = 100;   // a way to initialize inner member
```

VI. C++ Programming: Other features

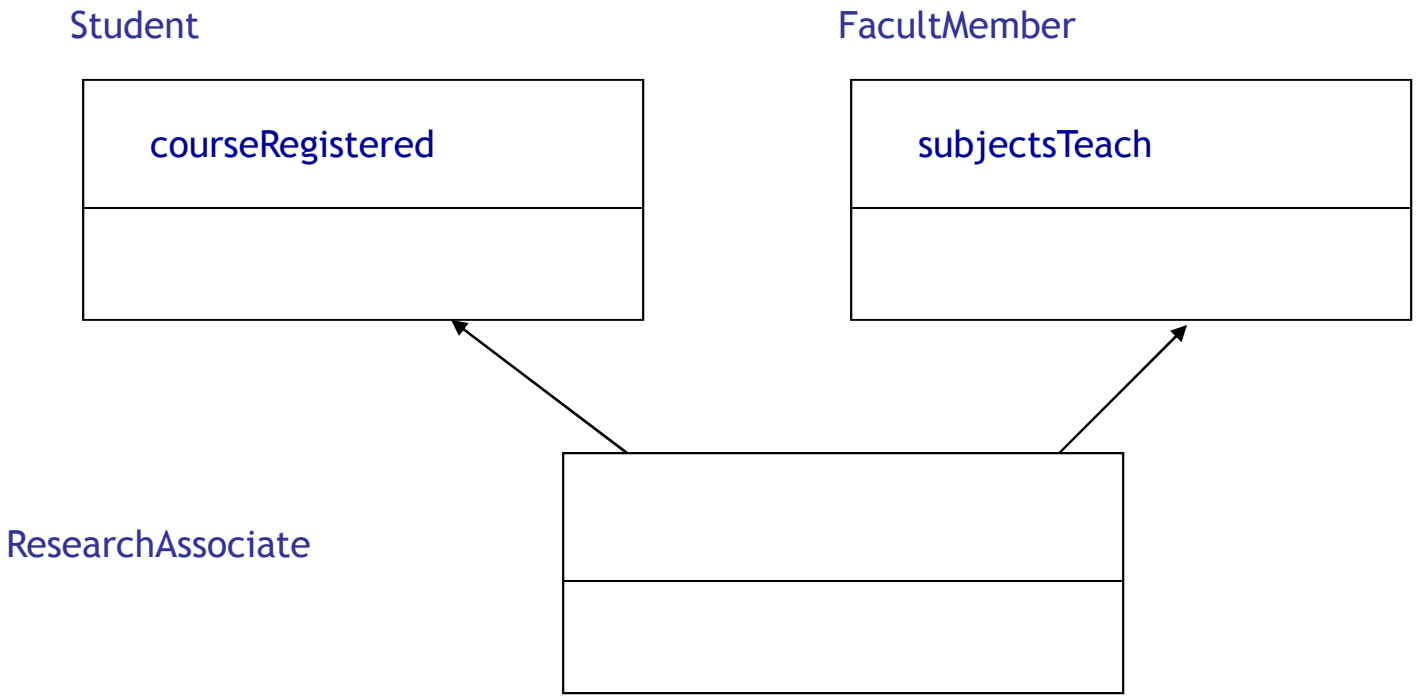
1. Nested Class

```
#include “InnerclassDemo.h”
#include <iostream.h>

int main( )
{
    Outer oObj;

    oObj.ShowOut( );

    return 0;

}
```

VI. C++ Programming: Other features

2. Other OO Modeling Concepts

- Multiple inheritance, is inheriting more than one class.



Research Associate in university is both a student (say, working on his Ph. D) as well as a faculty member (teaching at graduate level).

VI. C++ Programming: Other features

3. Inheritance – Aggregation – Association

- When to Use each of these relationships
  - Inheritance and Aggregation promote re-use of design and code.
  - Use inheritance based implementation when the real life has indeed a conceptual IS_A relationship.
    - e.g. Manager or Clerk IS first of all an EMPLOYEE
  - Use aggregation based implementation, when the real life has indeed a conceptual HAS_A or containment or assembly type of relationship. This is used when the contained object is not shared with objects of other (container) class, implying tight coupling.
  - e.g. CAR HAS WHEEL objects. Specific WHEEL objects will belong to only one CAR and wont be simultaneously related with other CAR.

VI. C++ Programming: Other features

3. Inheritance – Aggregation – Association

- When to Use each of these relationships
  - Use association relationship (or implementation), when the real life has a relationship implying a loose coupling.
    - e.g. EMPLOYEE works for DEPARTMENT.
  - Same department object is shared by all the employee objects.
  - Also, since this is a conceptual relation, it is possible to relatively easily disassociate an employee from one department and associate with other department. (unlike CAR and WHEEL relationship).

VI. C++ Programming: Other features

What we have covered

1. Nested Class
2. Other OO Modeling Concepts
3. Inheritance – Aggregation - Association

In this Module: What we have covered

I. OO Vs Functional approach
II. Introduction to Object Oriented Programming
III. Object Oriented Approach
IV. C++ NON Object Orientation features
V. Foundation of Object Orientation
VI. C++ Programming: Other features