Programming Paradigm

Note : These slides are not study material, rather just a guide to study these topics. Students are suggested to go through books and refer class notes to understand these topics in detail.

Introduction

Syllabus

1. **Programming Languages and Programming Paradigms**
2. **Imperative Programming**
3. **Functional Programming**
4. **Object Oriented Programming**
    - **C++**
    - **JAVA**
5. **Unified Modeling Language (UML)**

Computer Program

- A computer program (or just a program or software) is a sequence of instructions, written to perform a specified task with a computer.

- A program is expected to behave in a predetermined manner. No matter how many times one program is run, the same result should be received for same set of data provided

Programming Language

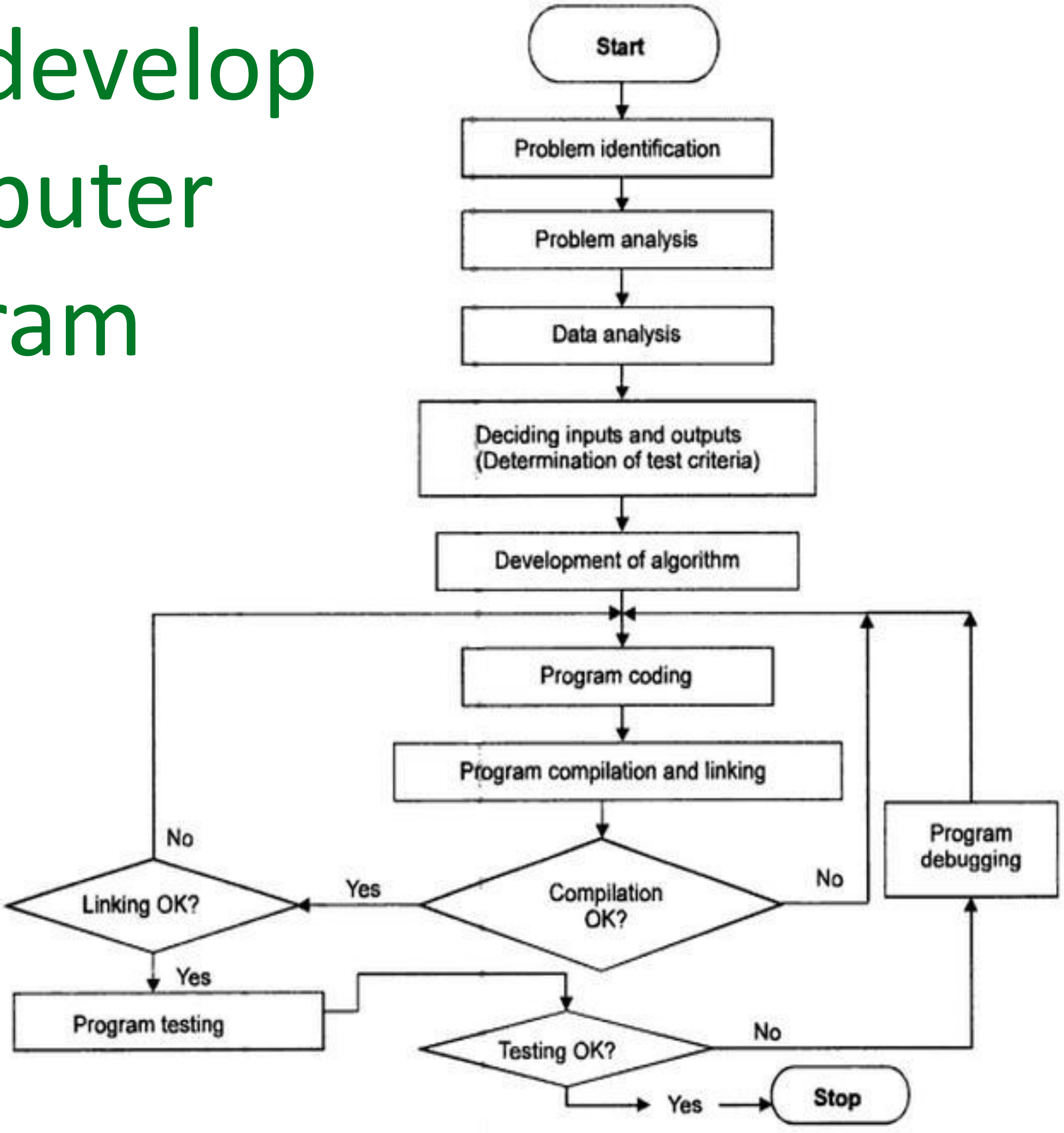
- Machine Language – Strict binary form / byte code
- High Level Language – C, Cobol, C++, Java, LISP etc
- Note : High level languages are compiled or interpreted to Machine language before execution

Computer Programming

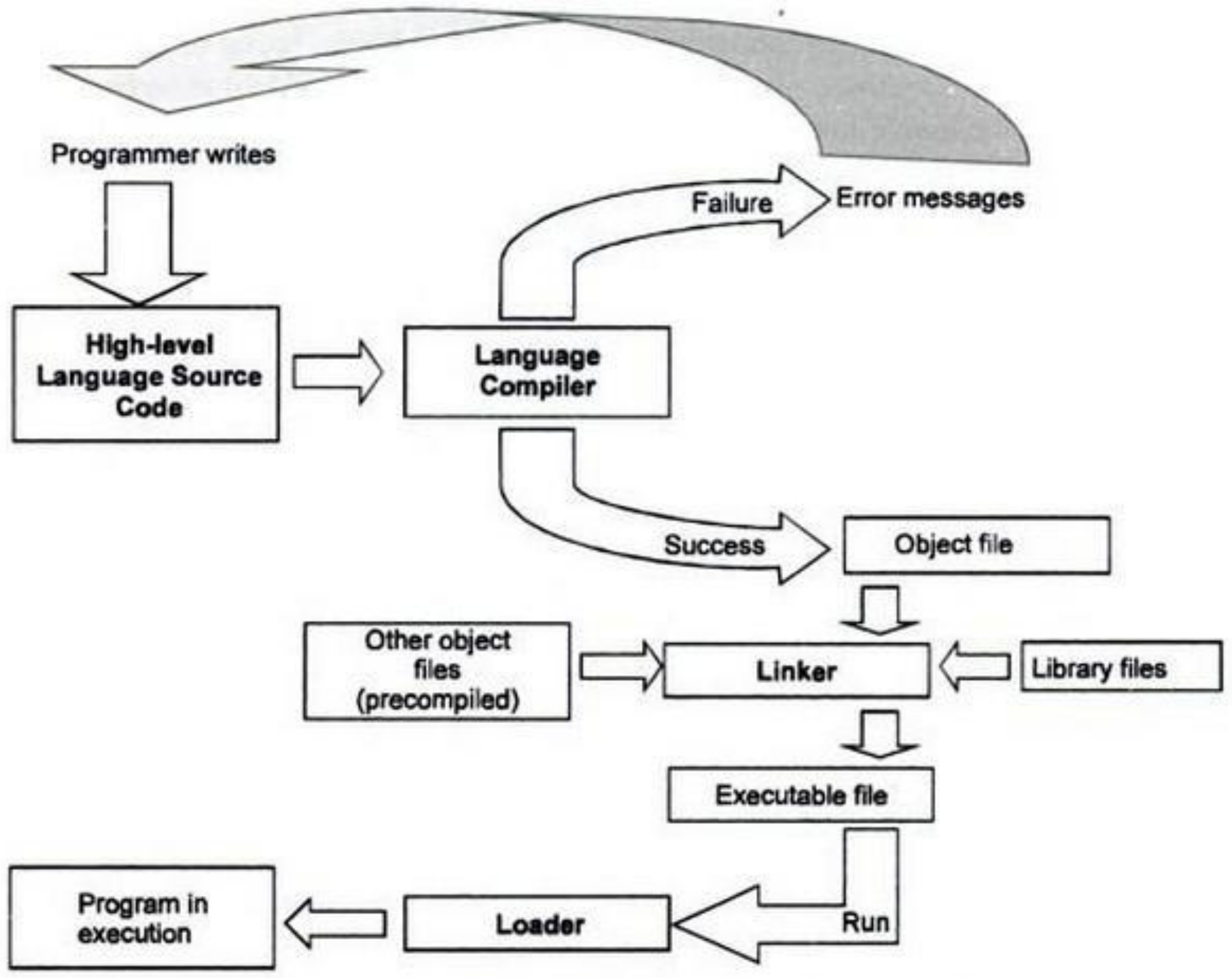
- Computer programming is the iterative process of writing or editing source code that can be executed in a computer

- It involves testing, analyzing, refining, and sometimes coordinating with other programmers on a jointly developed program

Steps to develop a Computer Program



Steps to execute a high level language program



Programming Paradigm

The basic structuring of thought underlying the programming activity

- Programming paradigm is a pattern or model of programming that derives the process of programming
- Every high level language has a paradigm that guides in a problem solving within a framework and gives a solution
  - It does not mean that all high level language are strictly following one particular programming paradigm
- Every Programming paradigm is a collection of conceptual patterns that control human thinking process to formulate the solution to a problem
- Different programming paradigms lead to different programming techniques

Programming Paradigm Contd…

Four main programming paradigms -

- **Imperative or Procedural** - Program as a collection of statements and procedures affecting data (variables). *FORTRAN*, *BASIC*, *COBOL*, *Pascal*, *C*

- **Functional** - Program as a collection of mathematical functions. *LISP*, *ML*, *Haskell*

- **Logic –** Program as a set of logical sentences. *Prolog*

- **Object Oriented -** Program as a collection of classes for interacting objects. *SmallTalk*, *C++*, *Java*

Brief on various Programming Paradigms

Imperative or Procedural

- Idea is - “First do this and next do that”, i.e. a step-by-step execution model - based on the stored program concept of Von Neumann
- Latin word "imperare" means “to command” - based on commands that update variables in storage
- It describes computation in terms of statements that change a program state.
- Similar to descriptions of everyday routines, such as food recipes
- Natural abstraction is function, procedures or subroutines

Imperative Programming (Example)

Sum of N positive numbers

```
Procedure sum(n)
Begin
  Define variable x with initial value of 1
  While the variable is greater than 1 do
  Begin
       Add x by n to store   result in x
       Decrement n
  End while
  Return value of variable x
end
```

Equivalent program in C

```
int sum(unsigned int n)
{
  int x = 1;
  while(n>1)
  {
      x += n;
      n--;
  }
  return x;
}
```

Functional Programming

- **Idea is** – *evaluation an expression and using the resulting value for something else*

- **Functions** are the fundamental building blocks (first class value) of a program. Functions in this sense (*<u>not to be confused with C Language functions which are just procedures</u>*) are analogous to mathematical equations: they declare a relationship between two or more entities.

- Based on mathematical model of function composition – *Lamda calculas*.

- The values produced are *non-mutable*

- No step by step execution model, result of one computation is input to the next and so on until some computation yields the desired result

Functional programming (Example)

Sum of N positive numbers

```
Sum n   = 1 (if n = 1)
          N + sum(n-1)
```

Equivalent program in LISP

```
(defun sum(n)
  (cond ((eq n 1) 1)
       (t (+ n (sum (- n 1))))
     )
)
```

Logic Programming

- It is based on the idea of answering a question through search for solution from a knowledge base
- Based on
  - **Axioms/Facts**
  - **Inferences rules**
  - **Queries / Goals**
- Program execution becomes a systematic search in a set of facts, making use of a set of inference rules – Set of know facts and set of rules results in deduction of other facts.
- Evaluation starts with a goal and attempts to prove it with a known fact or by deducing it from some rules.

Logic Programming (Example)

Axioms/Facts

```
F1. father(dasarath, ram).
F2. father(ram, lav).
F3. mother(kaushalya, ram).
F4. mother(sita, lav).
```

Inferences rules

```
R1. parent (X, Y) :- father (X, Y)
R2. parent (X, Y) :- mother (X, Y)
R3. grandfather(X, Y) :- father(X, Z), parent (Z, Y)
R4. grandmother(X, Y) :- mother(X, Z), parent (Z, Y)
```

Queries / Goals

```
G1. ? father(x, ram). X = dasaratha.

G2. ? father (dasaratha, X). X = ram

G3. ? grandmother(x, lav). X = kaushalya.

G4. ? parent (X, ram). X = ???
```

References

Book :

- C++ and Object-Oriented Programming Paradigm – Debasish Jana, PHI

Programming Paradigm

C++ Basic Concepts

Simple Output Operations

```
/* Demonstration of output operation */
#include <iostream>
int main(void){
    std::cout << “Hello from Amal” << std::endl;
    std::cout << “Hope you are ” << “doing well” <<std::endl;
    return 0;
}
```

Observe :

a) That the operator << can be used to place any simple type value (int, double, …) on to the variable of output stream

b) That the operator << has different semantics based on the context – this is also left shift operator on integer

c) That the result of the operation is a reference to an output stream variable – this allows cascading of the operator <<.

Simple Input / Output Operations

```
#include <iostream>
int main(void) {
    using std::cout;   using std::cin;    using std::endl;
    char name[20];  int age; char city[100];
    cout << "Greetings from Amal" << endl;
    cout << "Enter your name : ";
    cin >> name;
    cout << "Enter your age and city : ";
    cin >> age >> city;
    cout << "Hello " << name << ", you are " << age << " year old from " <<
   city << endl;
    return 0;
}
```

Observe :

a) Operator >> requires l-value or reference to a variable on the right

b) There is no necessity of finding the address of the variable

Namespace

When a big program split across files, it is possible that the variables introduced in the global scope may clash resulting in linker time errors.

To avoid the clashes of the names in the global space, names are introduced in one or more named space by the programmer.

There is a namespace called **std** to which the names of all standard library functions and other declarations are introduced

For your further study

Constant

- Constant in C++ have a name, type as well as a value.
- Definition of constant can be placed in `.h` files.
- Named constants are used in the program for the following reasons
  - Improves readability and therefore decreases the effort for maintenance
  - Can be modified in code at only one place

- Constants could be used -
  - Constants in program body
  - Constant parameter in function
  - Constant members in object / structure
  - Constant object through which a method is invoked

Constant

Example

```
int x; int y; int z;
const int a = 10;    // a is a constant integer
// const int b;      // syntax error - no value provided

int * const p = &x;  // p is a const pointer to integer
                     // should be initialized with definition
// p = &y;           // value of p can NOT be changed
*p = z;              // value of *p can be changed

const int * q = &x;  // q is a pointer to an int which is a const
q = &y;              // value of q can be changed
// *q = z;           // value of *q can NOT be changed
```

Reference Variable

Example code :

```
int a = 10; int x = 30;
int &b = a; // b is a reference to a; b is same as a.
            // no new integer location is allocated to b
            // whatever b is used, it automatically refers to a

cout << &a << " " << &b << endl;  // outputs same address
cout << b << endl; // outputs 10

int c = b; // c becomes 10
cout << c << endl; // outputs 10

c = 20;  // b remains unchanged
cout << b << endl; // still outputs 10
```

Reference Variable

Contd...

Example code (continued from previous slide):

```
b = c; // same as a = c
cout << a << endl; // outputs 20
// int &d; // Syntax error
// &b = x; // Syntax error

// int & arr[] = {a, x}; // Array of reference are not allowed
```

Few points on Pointers & Reference

- Pointers may be undefined or NULL; reference should always be associated with a variable

- Pointers may be made to point to different variable at different time; reference is always associated with the same variable throughout its life

- Pointers should be explicitly dereferenced; References are automatically dereferenced

Pointers and dynamic allocation

- C++ supports dynamic allocation (and deallocation) through two operators : **new** and **delete**
- The operator **new** requires a type as the operand. It allocates as much memory as required for the particular type and returns a pointer to the type specified
- The operator **new** also works on user defined types
- The operator **new** and **delete** can be overloaded (will be discussed later)

```
Example code :
    int *p = NULL;
    p = new int;
    *p = 20;
    delete p;
    int *q = new int[5];
    for(int i = 0; i < 5; i++) {
        q[i] = i; }
    delete []q;
```

Remembering Alias

Alias example code

```
    int *p, *q;
    p = new int;
    *p = 20;
    q = p; // q and p refer to the same location, they are now
aliases
    *q = 30; // even *p would have changed
```

Remembering Garbage

Garbage example code

```
    int *p = Null;
    p = new int;  // Space for an int is create/reserved
    *p = 20;
    p = new int; // Space for one more int is created, earlier
value and location lost - it's a memory leak and also
called garbage location without accesses
```

Remembering Dangling reference

Dangling reference example code

```
    int *p, *q;
    p = new int;
    *p = 22;
    q = p; // q and p refer to the same location, they are now
aliases
    delete p; // space for int is released
    *q = 333; // The pointer q points to a non-existing
location. This is called dangling reference. This can
causes memory corruption
```

Function Overloading - Static Polymorphism

It is possible to have more than one function with the same name where the calls are resolved at compile time based on matching the arguments to the parameters

Points on function overloading :

- More than one function with the same name
- Function call resolution is a compile time phenomenon and there is no extra overhead at runtime
- Functions should be declared in the same scope
- Resolution based on the number, type and order of arguments in the function
- Resolution does not depends on the result type

Function Overloading

An example

```
#include<iostream>
using namespace std;
void display(char * name) {
    cout << "Name " << name << endl;
}
void display(char * name, int age){
    cout << "Name " << name << " Age " << age << endl;
}
int main() {
    char name[10] = "Aman";
    int age = 10;
    display(name, age); // Outputs : Name Aman Age 10
    display(name); // Name Aman
    return 0;
}
```

Default Parameter

There are many functions which have a large number of parameters. The user may not be interested in providing all these arguments in the call and he might be happy if the system can choose some default appropriate values for some of these parameters.

Example code

```
void display(const char * = "India", int = 66);
int main() {
    display("Bangladesh", 42); // Calling with two arguments
    display("India"); // Calling with one argument
    display(); // Calling with no argument
    return 0;
}
void display(const char * name, int age) {
    cout << "Country " << name << " Age " << age << endl;
}
```

Default Parameter

Few points

- The parameters should have a reasonable default
- It is a way of expanding the function
- Default value(s) are specified as part of the declaration
- Can be specified for functions whose source codes are not available – function writer may not know anything about these defaults
- Only rightmost parameters can be default – all parameters can also be default
- Leftmost arguments should be specified

Call by Reference or Reference Parameters

- '**C**' provide only one mechanism of parameter passing – by **Value**.
  - Arguments are copied to the parameters and changes to the arguments are not reflected in the calling functions

  If arguments should be changed by the called function, then the pointer (<u>that is also value</u>) to the arguments is passed. Thus simulating parameter passing by reference in C (but its is just a simulation).
  - In the called function, pointer has to be dereferenced to access the arguments. This is costly operation.

- 'C++' supports parameter passing by **Value** as well by **Reference**.
  - As reference parameters are just a copy of the actual parameter, there is no need of dereferencing, Reference parameters are automatically dereferenced.

Call by Reference

Example

```
void swap1(int x, int y) {
    int temp; temp = x; x = y; y = temp; }
void swap2(int * x, int * y) {
    int temp; temp = *x; *x = *y; *y = temp; }
void swap3(int & x, int & y) {
    int temp; temp = x; x = y; y = temp; }

int main() {
    int a = 10, b = 20;
    swap1(a, b); // call by value - changes not reflect
    swap2(&a, &b); // call by value with pointer - changes reflects
    swap3(a, b); // call by reference - changes reflects
    return 0;
}
```

Call by Reference

Few points

- Less overhead in terms of time & space at runtime compared to parameter passing by value
- *Can avoid problems resulting in* ***shallow*** *copy of parameter passing by value – this happens when a structure or an object has a pointer as a member*
- Can be passed `const` reference to avoid changing the argument in the function – this is in case arguments are not required to change in the the called function
- It is also possible to return by reference when an lvalue is required in the calling function
- It is efficient w.r.t. passing by pointer for a complex structure as dereferencing is not required

Conclusion : Always preferable to pass by reference

Inline function

- Inline functions are used to reduce the overhead of normal function call
- It's a request to the C++ compiler that inline substitution of the function body is to be preferred to the usual function call implementation
- Suggestion could be ignored based on the compilers discretion
- Unlike Macro, it does not leave any side effect due to expansion of code
- Drawback is source code has to be part of client program – information hiding and encapsulation suffers a serious setback

Example Code :

```
–   #define MC_SQUARE(X) X*X
   inline int fn_square(int x) { return x*x;  }
   int a = b = 3;
   int i = MC_SQUARE(++a);    // results in 20
   int j = fn_square(++b);    // results in 16
```

Template function

There are many instances where the structure of the code (logic) does not change with the input type. In such case it is possible to write generic routines where the type information can be passed as parameter in the calling code.

Example code :

```
template <typename T>  // T is a type parameter
void add(T &x, T &y) {
    T temp = x + y; cout << temp << endl;
}
int main() {
    int a = 10, b = 20;
    double p = 2.4, q = 4.5;
    add(a, b); // Causes compilation type instantiation of the
               template function with type T = int.
    add<double>(p, q); // Another instance of the template
                       function with type T = double.
}
```

Template function

Few points

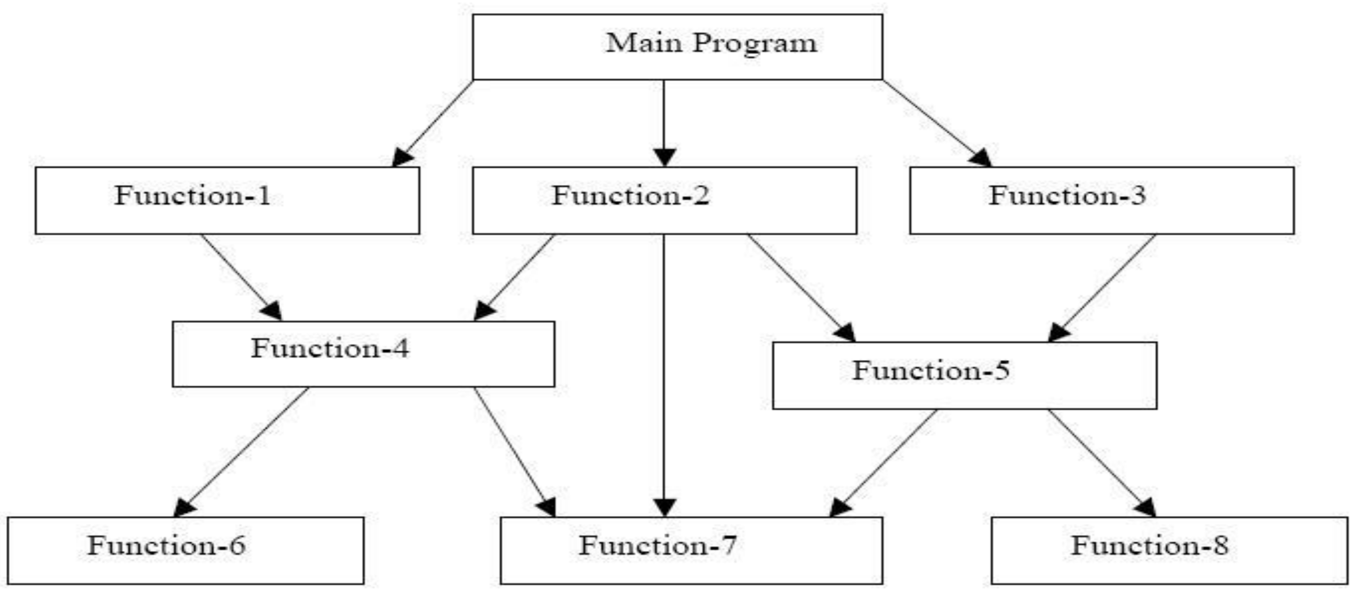
- Mechanism to generate functions at compile type
- No extra overhead at run time
- Used to making functions generic
- Used when behavior/algorithms of the functions are same
- Gets implicitly instantiated by the call
- Can also be explicitly instantiated
- Instantiated for each type only once

Object Oriented Paradigm

Object Oriented Concepts

Procedural Programming

- The problem is viewed as the sequence of things to be done such as reading, calculating and printing
- The primary focus is on functions - the technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem



Procedural Programming Contd…

- In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions.
- Global data are more vulnerable to an unintentional change by a function.
- In a large program it is very difficult to identify what data is used by which function.
- In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.
- *Can not model real world problems very well - functions are action-oriented and do not really corresponding to the element of the problem*

Procedural Programming Characteristics

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

Object Orientation

The Big Picture

- Object Orientation is a programming paradigm - it is not a computer language itself, but a set of ideas those supported by several languages

- The problem-solving techniques used in object-oriented paradigm is more closely models the way humans solve day-to-day problems

- An object-oriented program is structured as community of interacting agents called objects.
  - Each object has a role to play.
  - Each object provides a service or performs an action that is used by other members of the community

Object Orientation

An example

Suppose you wanted to send flowers to your **Mother** who lives in another city. To solve this problem you simply walk to your nearest **florist** run by, lets say, **Raman**. You tell **Raman** the kinds of flowers to send and the address to which they should be delivered. You can be assured that the flowers will be delivered.

Object Orientation

An example Contd…

Lets examine the mechanisms used to solve your problem

- You first found an appropriate **agent** (Raman, in this case) and you passed to this agent a **message** containing a request.
- It is the **responsibility** of Raman to satisfy the request.
- There is some **method** (an algorithm or set of operations) used by Raman to do this.
- You do not need to know the particular methods used to satisfy the request - such information is **hidden** from view.

This leads to our first conceptual picture of object-oriented programming:

"*An object-oriented program is structured as **community** of interacting agents called **objects**. Each object has a role to play. Each object provides a **service** or performs an action that is used by other members of the community.*"

Object Orientation

Messages and Responsibilities

Members of an object-oriented community make requests of each other.

- Action is initiated in object-oriented programming by the transmission of a **message** to an agent (an **object**) responsible for the actions.

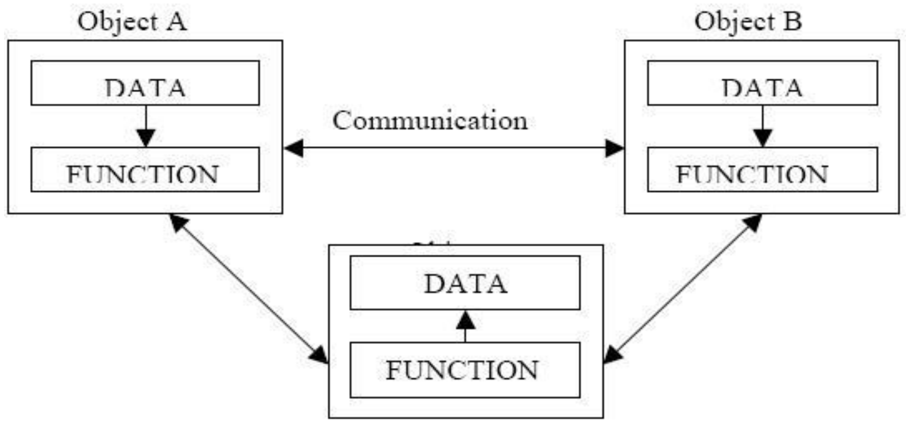
- The message encodes the request for an action and is accompanied by any additional information (arguments/parameters) needed to carry out the request.

- The **receiver** is the object to whom the message is sent. If the receiver accepts the message, it accepts **responsibility** to carry out the indicated action.

- In response to a message, the receiver will perform some **method** to satisfy the request.

Object Oriented Paradigms

- OOP treats data as a critical element in the program development and does not allow it to flow freely around the system.
- It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function.
- OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects.
- The data of an object can be accessed only by the function associated with that object
- One object can communicate to another object using functions associated with that object (message passing)



Object Oriented Programming Characteristics

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects. Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are ties together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

Object and Class concept

Object

An object represents a tangible entity

- Physical
- Conceptual

Object are always <u>not</u> **physical** items and <u>not</u> **visible** items

Object orientation in computing is intended to thinking about programming closer to think about real world.

"What is an object in computer program => What is the object in real world ?"

Object : A more formal definition

- An object is an entity that has a well-defined boundary. That is, the purpose of the object should be clear.

- An object has three key components :
  - **Identity** – Distinguishes one object than other
  - **Attributes** – Information describes their current state
  - **Behaviour** – Specific to an object

- Object in a computer program are self contained. They are independent to each other.

An Object has State

- The state of an object is one of the possible conditions that an object may exists in and it normally changes over time.

- The state of an object is usually implemented by a set of properties call attributes. Most object can have multiple attributes

- State of one object is independent than another

An Object has State

Contd...

PROFESSOR		PROFESSOR
Name : J Clark		Name : J Clark
Employee Id : 56789		Employee Id : 56789
Date Hired : 25/07/1991	$\longrightarrow$	Date Hired : 25/07/1991
Status : Tenured		Status : Retired
Decipline : Finance		Decipline : None

An Object has Behaviour

- Objects are intended to mirror the concepts that they are modelled for
- Behaviour is specific to the type of the object
- Behaviour determines how an object act and react to request from other objects
- Object behaviour is represented by the operations that the objects can perform.

An Object has Behaviour

Contd…

PROFESSOR J Clark's behavior

Take Class Tests

Submit Final Grade

Accept Course Offering

Take Sabatical

An Object has an Identity

- The term **Identity** means that objects are distinguished by their <u>inherent existence and not by descriptive properties </u>at they may have

- The same concept holds true for objects. Although two objects may share the same state (attributes), the are separate, independent objects with their own unique identity

- One object may contain another objects that, still they are separate.

An Object has an Identity

Contd...

Professor : Clark

Teacher Biology

Professor : Clark

Teacher Biology

Computing Object

Objects in real world generally we consider objects those we can to see or touch.

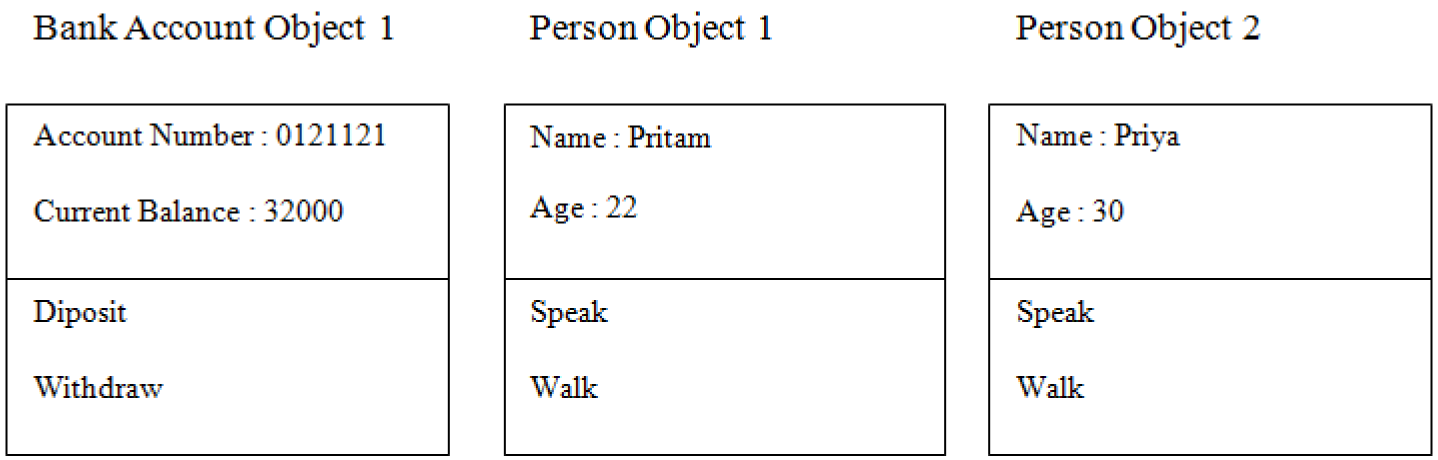
But in computing we take it further – Date, Time, link-list, Bank Account, Event can be objects. These are well defined idea.

Object are always <u>not</u> **physical** items and <u>not</u> **visible** items

Computing Object Contd...

Object in a computer program are self contained. They are independent to each other.

Like real world Object, they have identity, state and behaviour



Bank Account Object 1

Account Number : 0121121

Current Balance : 32000

Diposit

Withdraw

Person Object 1

Name : Pritam

Age : 22

Speak

Walk

Person Object 2

Name : Priya

Age : 30

Speak

Walk

How to find Object in computing

For those are new in Object Oriented Design, it could be potentially challenge to decide on objects in a compute program from the problem definition

It's easy to decide when a computer program need to objects like Car, Employee, document etc.

But while developing application like Event Management application, what about an Event ? Is an “Event” be an **o**bject ?

How to find Object in computing

Contd ...

Basic clue :

In the problem statement check for the the words those are noun (physical entity or idea or concept) and add “the” in front those - like the television, the car, the bank account, the time, the event

And verbs in the problem statement are behaviour associated to those Object – like flying, printing, exploding etc.

Where do the Objects come from ?

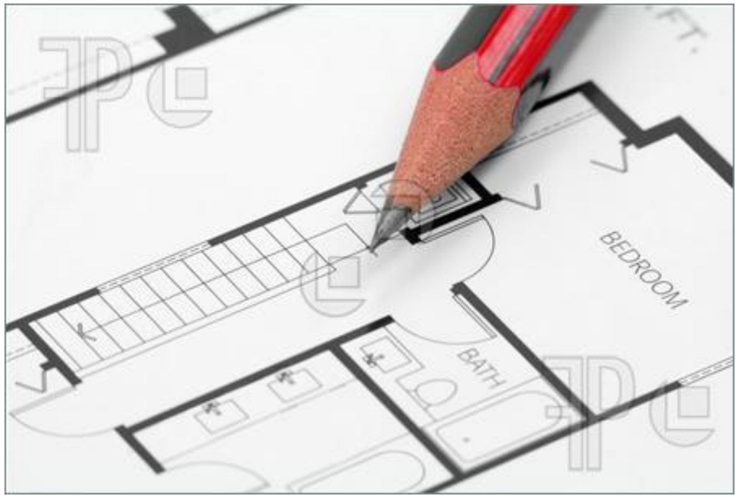
There is another word, that goes hand-in-hand with object - that word is **Class**

Entire point in OOD, is not about Objects, rather Class

In Computer program Class comes first, NOT the Object

Class

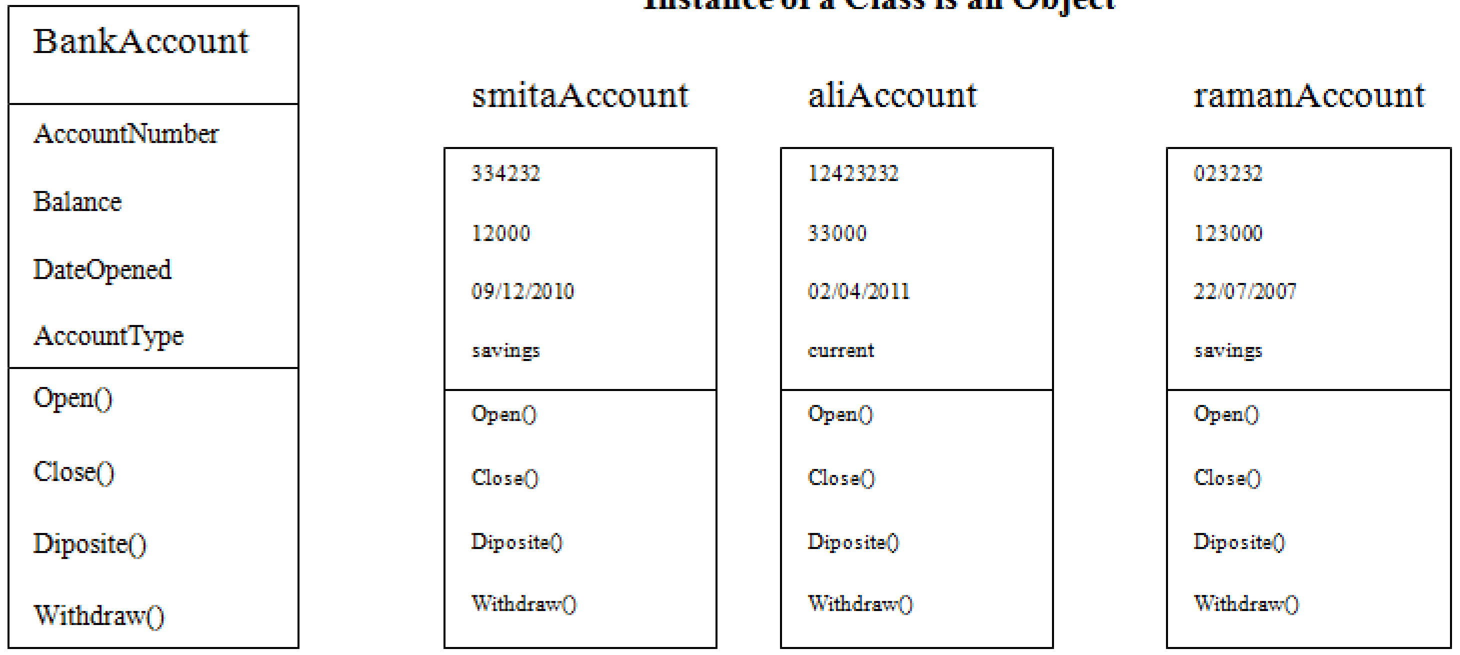
- A Class describes what an Object will be, but it is not the object itself
- A Class is a blue print – the detail description and definition
- *A Class is a description of a set of objects that share the same properties(attributes), behaviour(operations), kind of relationships, and semantics.*



Class / Object

Creating Objects = Instantiation

Instance of a Class is an Object



Class **Objects**

Existings Classes in OOP languages

- Most OOP languages provide many pre-written generic classes at minimum : for example string, dates, file I/O, networking – so sometime you can start creating objects without writing classes
  - ➢ C++ standard Library
  - ➢ Java Class Library – more than 4000 classes available
  - ➢ .Net standard Library – more than 4000 classes available
  - ➢ Ruby Standard Library
  - ➢ Python Standard Library
- But in non-trivial application, you still need to write Classes following few principle of OOP ...

Fundamental OOP concepts

Fundamental of OOP Concepts

<u>**A**</u>bstraction

<u>**P**</u>olymorphism

<u>**I**</u>nheritance

<u>**E**</u>ncapsulation

Abstraction

Abstraction means focusing the essential quality of something, rather than one specific example.

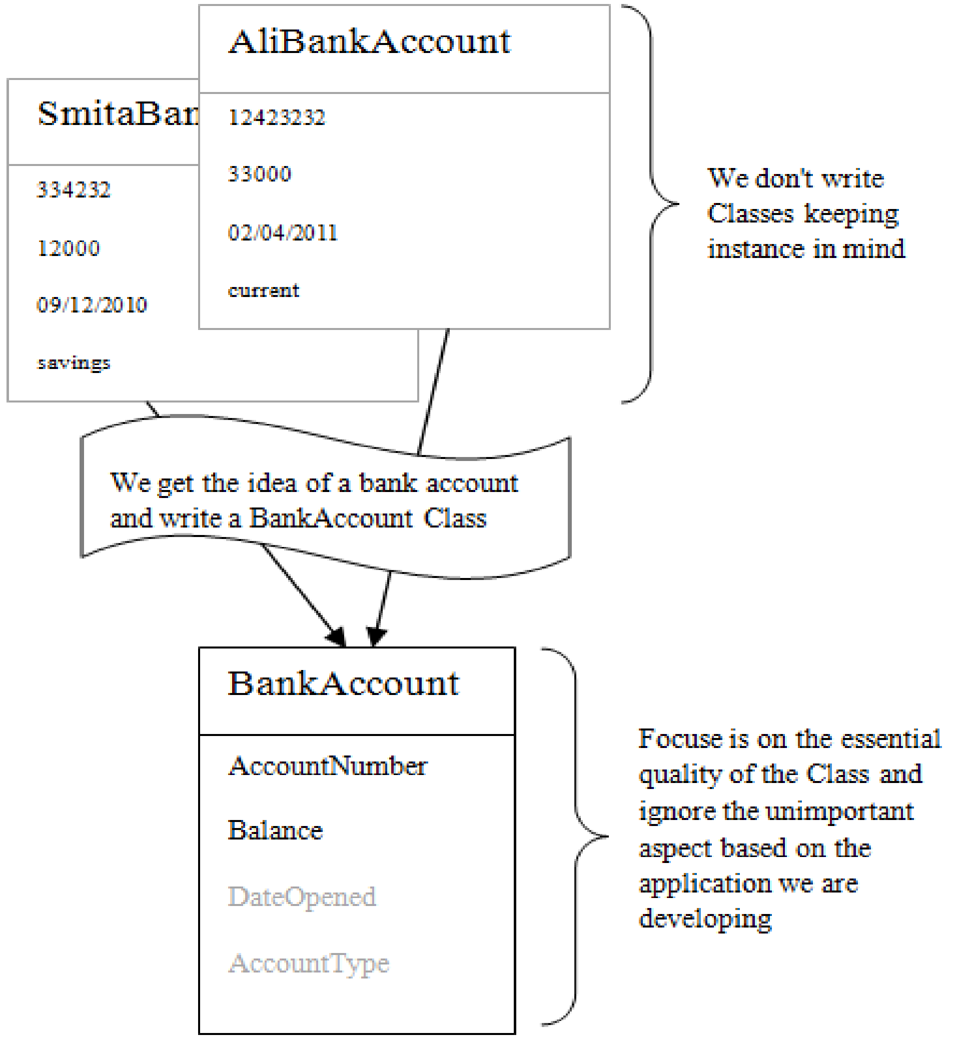
It means automatically discard the unimportant and irrelevant.

<u>Abstraction can be defined as :</u>

"Any model that includes the most important, essential or distinguishing aspects of something while suppressing or ignoring less important, immaterial or diversionary details."

Abstraction

Example



AliBankAccount
12423232
33000
02/04/2011
current
SmitaBan
334232
12000
09/12/2010
savings
We don't write Classes keeping instance in mind
We get the idea of a bank account and write a BankAccount Class
BankAccount
AccountNumber
Balance
DateOpened
AccountType
Focuse is on the essential quality of the Class and ignore the unimportant aspect based on the application we are developing

Abstraction Points to remember

- Abstraction is as an external view of the object.
- Abstraction allows us to manage complexity by concentrating on the essential characteristics of an entity that distinguishes it from all other kind of entities.
- An abstraction is domain and perspective dependent. That is, what is important in one context, may not be in another.
- In the context of OOP, abstraction of an object means its outside view provided by the interface.
    - Note that the interface only tells WHAT the object can do, it does not says HOW it performs its work

Encapsulation

Encapsulation means hiding implementation detail from external world.

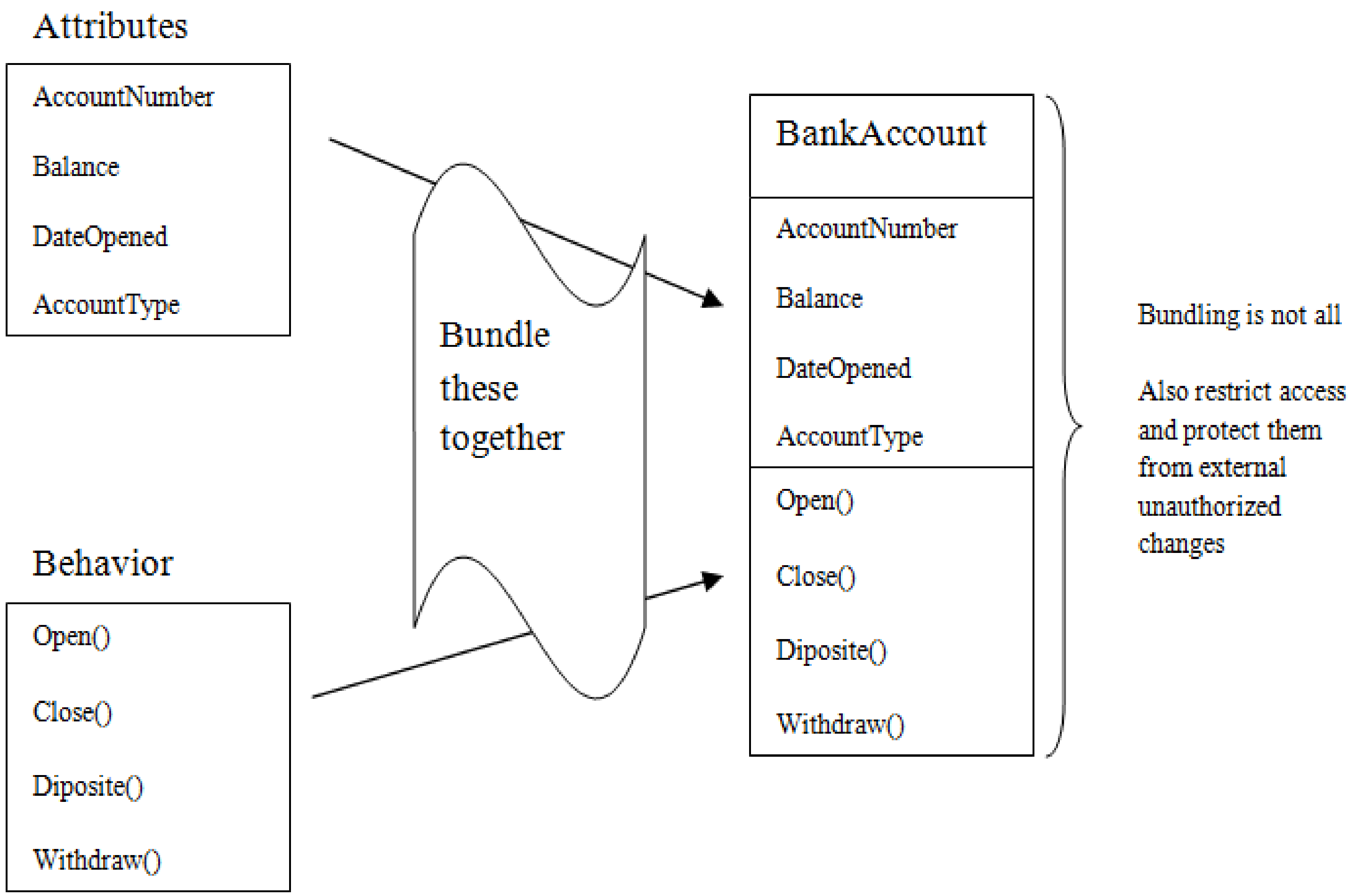
It is the idea of surrounding something, not just the keep the content together, but also to protect the content

<u>Encapsulation can be defined as :</u>

“The physical localization of features (e.g. properties, behaviors) into a single blackbox abstraction that hides their implementation (and associated design decissions) behind a public interface”

Encapsulation

Example



Bundling is not all

Also restrict access and protect them from external unauthorized changes

Encapsulation Points to remeber

- It is the approach of putting related things together
- Distinguishes between interface and implementation – exposes interface, hides implementation
- Encapsulation is often referred to as “information hiding”, making it possible for the clients to operate without knowing how the implementation fulfills the interface
- It is possible to change the implementation without updating the clients as long as the interface is unchanged

Inheritance

Inheritance is the process of involving building classes upon an existing classes. So that additional functionality can be added.

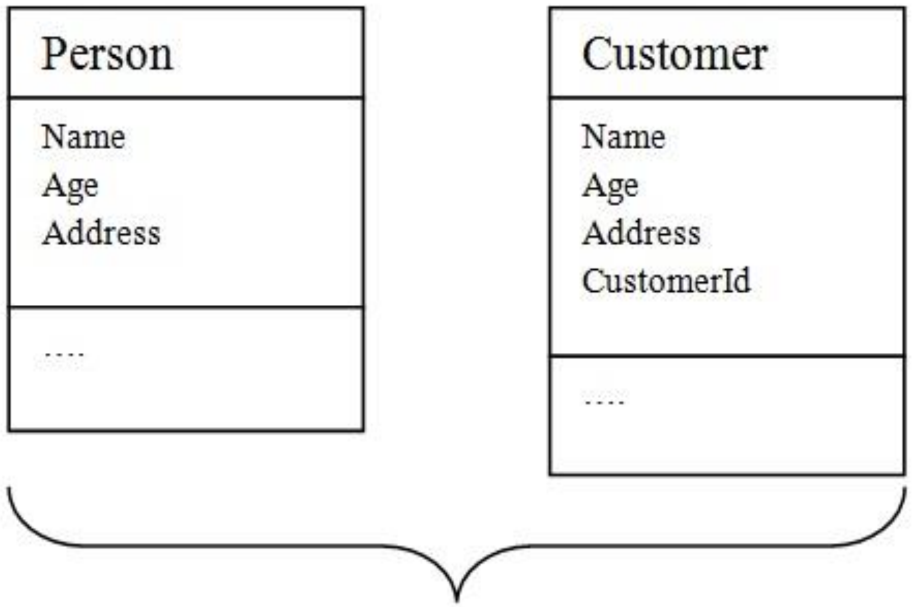
Using inheritance the hierarchical relationships are established. Inheritance allows the reusability of an existing operations and extending the basic unit of a class without creating from the scratch.

<u>Inheritance can be defined as :</u>

"*Inheritance is a way by which a newly defined Class inherits attributes and behaviors of an existing Class along with its own properties*"

Inheritance

Example



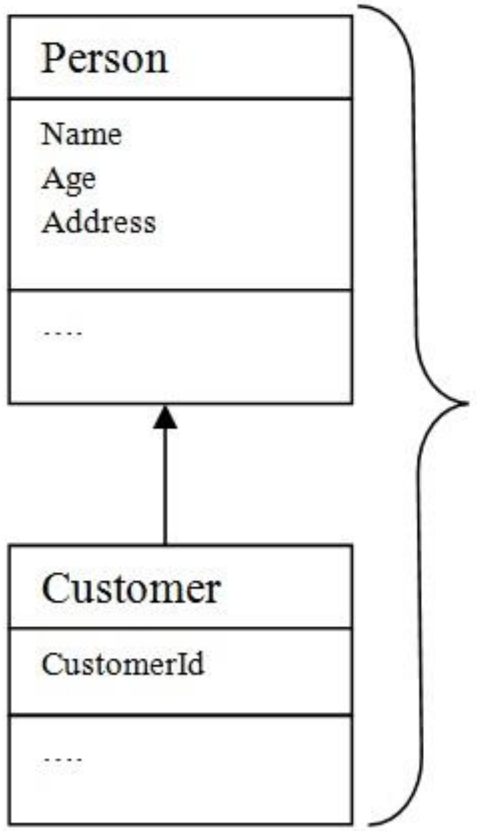
Customer class is exactly the same as Person class with some extra Attributes and Behavior.

Approaches could be -

- Writing the entire new class for Customer – but reusability is not achieved
- Add additional property to the Person Class –the Abstraction property for Person class gets violated

Inheritance

Example Contd...I



Customer class automatically has everything (such as Attributes and Behavior) in Person class without writing any code.

As well it has its own specific property

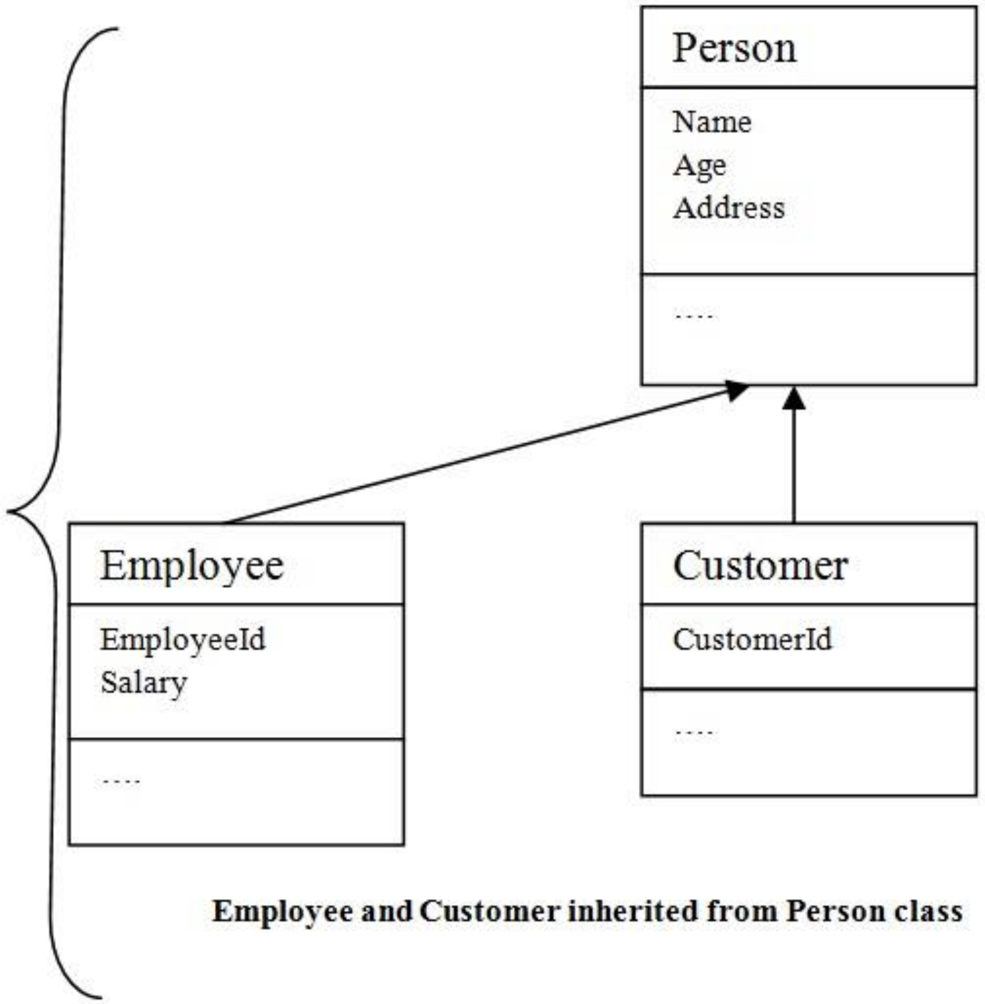
It's a "is a" relationship

Customer inherited from Person class

Inheritance

Example Contd…II

Multiple classes can be inherited from one class. Both the derived class can co-exists without impacting each other.



Employee and Customer inherited from Person class

Base Class	Parent Class	Super Class
Derive Class	Child Class	Sub Class

Inheritance Points to remember

- Its is a “is a” relation between Classes
- Used for creating hierarchy of types
- Its a code re-use mechanism
- One Base Class can be inherited by multiple Derived Classes
- One Derived Class can inherit properties from multiple Base classes – Multiple inheritance

Polymorphism

The Greek term polymorphism means "*having many forms*" - means being able to invoke different kinds of functionalities using the same interface"

Polymorphism can be defined as :

"Polymorphism is sharing a common interface for multiple types, but having different implementation for different types"

In OOP, polymorphism is a technique where objects of classes belonging to the same hierarchical tree (i.e. inherit from a common base class) and may possess interface bearing the same name but each having different behaviors

Polymorphism

Analogy

A + B = ?   What + sign do ?

For some language -

- When A and B are **Integer** variable having value 6 and 7, it results 13.

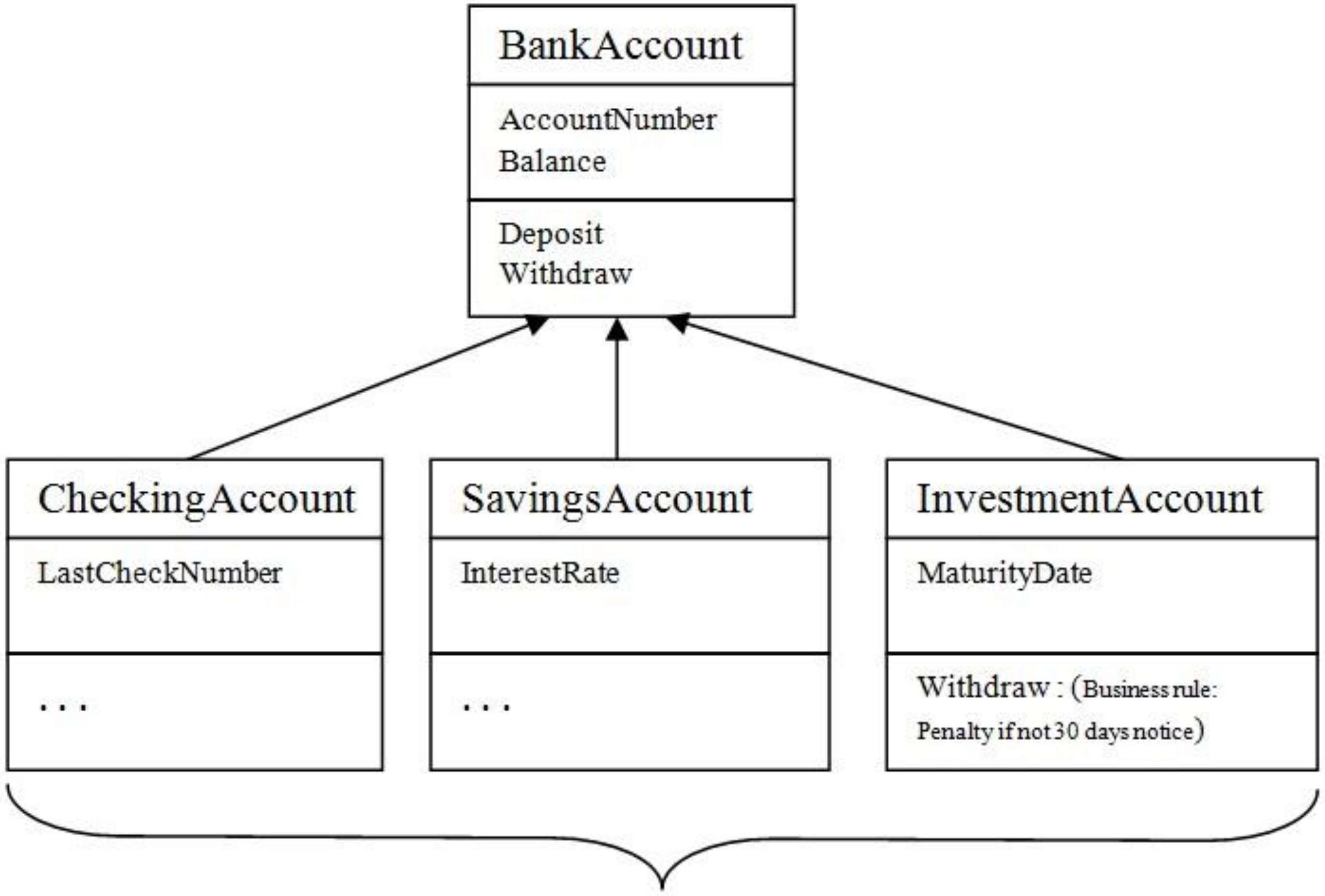
  The + sign behaves as **Addition**

- When A and B are **String** variable having value “Hello” and “World”, it results “HelloWorld” .

  The + sign behaves as **Concatenation**

It does automatically correct and different behavior based on the context of operation – this is the philosophy of Polymorphism

Polymorphism



Withdraw behavior is written differently for this particular Class – this is **overriding**.

Rule: Inherit when useful, Override when not useful

Lets assume there is array of 10K account Objects of different types loaded in the computing system. Now Withdraw method can be called each one of these Objects without knowing the type of the Class. Withdraw method will do the correct behavior for each one – this is called Polymorphic behavior

Polymorphism Points to remember

- Its the way of inheriting when useful, overriding when not useful
- It allows automatically do the correct behavior even if we are working with many different forms
- There are two kinds of Polymorphism
  - Static Polymorphism
  - Dynamic Polymorphism

Programming Paradigm

Object Oriented Paradigm : Class and Object using C++

Structure

In “C”; it is possible put related items together, even if they are heterogeneous, in a structure.

Structures thus encapsulate related data, but it does not provide a mechanism by which we can also specify how the data can be acted upon

```
struct Person {
    char name[20];
    int age;
}
void Read(Person &x){
    cout << “enter name and age
:”;
    cin >> z.name >> z.age;
}
```

```
void Write(const Person & y) const
{
    cout << “Name : “<< y.name << “
age : “ << y.age << endl;
}
int main() {
    Person p;
    Read(p);
    p.age = -10; // it is possible
    Write(p)
    return 0;
}
```

Structure

Contd...

In "C++"; it is possible put not only data, but also operation with in a structure. It is new concept in C++

But in structure, by default all members are **Public** and therefore can be access by client program.

```
struct Person {
    char name[20];
    int age;
    void Read();
    void Write() const;
}
void Person::Read(){
    cout << "enter name and
age :";
    cin >> name >> age;
}
```

```
void Person::Write() const {
    cout << "Name : "<< name << " age
: " << age << endl;
}
int main() {
    struct Person p;
    p.Read();
    p.age = -10; // Still it is
possible
    p.Write()
    return 0;
}
```

Class

The extended feature of structure in C++, we would prefer to call as Class where we can have the **Access Specifier**.

Note that in Class, by default all members are **Private**

```
class Person {
    private : // Even if not specified
        char name[20];
        int age;
    public :
        void Read();
        void Write() const;
}
void Person::Read(){...}
void Person::Write() const {...}
```

```
int main() {
    Person p;  // p is an object of the class Person
    p.Read(); // p implicitly passed by reference

    p.age = -10; // It is NOT possible

    p.Write() // p is passed as a constant object
    return 0;
}
```

Class

Few points to note

<u>Class :</u>

- Encapsulates data and functions
- Same as structure, but all the members are private by default
- An object is an instance of the Class
- Size of the object depends only on the data members of the Class ad their layout and does not depends on the member functions
- Invoking member functions of the class is resolved at compile time and therefore no extra overhead at run time

Access Specifier

Private : default specifier for the class, can be access only by member function of the class or friend function (*This point will be revisited while discussing friend function*).

Public : can be access from the client function also

Protected : can be access from member function of the same class or any of it's publicly derived class – can not be access by the client function (*This point will be revisited while discussing inheritance*).

'This' pointer

The keyword ***this*** identifies a special type of pointer.

If an Object of a Class is created that has a non-static member function -

- When the non-static member function is called – the keyword ***this*** in the function body stores the address of the Object (acts as a stack variable).
- When a non-static member function is called, the ***this*** pointer is passed as an extra argument (hidden).

Its not possible to declare the ***this*** pointer or make assignments to it

A static member function does not have a this pointer

'This' pointer

Example

```
class Person {
    private :
        char name[20];  int age;
    public :
        void Read();
        void Write() const;
}
void Person::Read(Person * this){
    cout << "enter name and age :";
    cin >> this->name >> this->age;
}
void Person::Write(Person * const this) const {
    cout << "Name : "<< (*this).name << " age : " << (*this).age <<
endl;
}
```

Default member functions of a Class

What are the member-functions a class has by default ?

By default, if not implemented by the user, the compiler add some member functions to the class. Those are below four -

- Default Constructor
- Destructor
- Copy constructor
- Assignment operator

Note : This is true with some exceptions

Constructor

An object of the class can be initialize by special function called the constructor.

<u>Constructor :</u>

- Is a member function which is used to initialize the object
- Has the same name as the class
- Is invoked when an object is created
  - Is <u>NOT</u> invoked when a pointer of an Class is defined
  - Is invoked when an object is created dynamically
  - Is invoked as many times as the member of the elements in an array of objects
- Has no return type
- Can have parameter and default parameter
- Can be overloaded
- A constructor which takes no arguments is called **Default Constructor**
- A constructor can be private (will be discussed later)

Destructor

Any clean-up operation of an object can be done by another special function called the destructor.

Destructor :

- Is a member function which is used to release resources
- Has the same as the class, preceded by ~
- Is invoked when an object is removed
    - Not invoked when a pointer of a Class goes out of scope
    - Invoked when a dynamically allocated object is deleted
    - Invoked as many time as the number of elements in an array when array goes out of scope
- Has no return type
- Can have no parameters
- Can not be overloaded

Constructor and Destructor

Example

```
class Person {
    private :
        char name[20];
        int age;
    public :
        Person(); // default
        Person(char *,
int=20);
        ~Person();
        void Read();
        void Write() const;
}
Person::Person(){
    name[0] = '\0'; age = 0;
}
```

```
Person::Person(char *s, int n){
    strcpy(name, s); age = n;
}
Person::~Person(){
    cout << “destructor called” <<
endl;
}
int main() {
    Person p;
    Person q(“Raman”, 53);
    Person * ptr = Null;
    ptr = new Person(“Sundar”);
    delete ptr;
}
```

Initialization List

Initialization lists are used to initialize the data members of the class

They can be used only in the constructor and no other member function

Initialization list is executed before the body of the constructor is executed

```
Example code :
Person::Person() : age(0) {
    name[0] = '\0';
}
Person::Person(char *s, int n) : age(n) {
    strcpy(name, s);
}
```

Initialization List

Few points

- Used in constructor to initialize data members of the class
- Invoked in the order of declaration in the class and not in the order of the occurrence in the initialization list
- Efficient compared to making assignment
- **Necessary to initialize constant or reference data member of the class**

<u>Below points will be revisited while discussing inheritance</u>

- Necessary to invoke the constructor of the base class
- Base class constructors are invoked in the order of derivation and not in the order of occurrence in the initialization list
- If a base class constructor is not specified, default base c;ass constructor will be invoked
- Base class constructor are invoked before the members of the class are initialized

Dynamic Memory allocation using Constructors and Destructors

```
class Person {
    private :
        char * name; // a pointer
        int age;
    public :
        Person(); //default
        Person(char *, int);
        ~Person();
        void Read();
        void Write() const;
}
Person::Person(): name(NULL),
age(0) { }
```

```
Person::Person(char *s, int
n):age(n){
    name = new char[strlen(s)+1];
    strcpy(name, s);
}

Person::~Person(){
    delete []name;
}

int main() {
    Person p;
    Person q("Raman", 53);
}
```

Copy Constructor

There are instances in the code where an object is initialize with an existing object.

```
Example code :
int main() {
    Person p("raman", 53);
    p.write();
    Person q(p); // q is a new object instantiated by an existing
  object p, equivalent to "Person q = p;"
    q.write();
}
```

In such case, a special constructor called the copy constructor gets invoked.

A default copy constructor provided by the compiler that does a member wise copy (**shallow copy**). This will not work if the object has a pointer.

In that case programmer is required to provide an implementation of copy constructor if the object has some resource (for example : pointer data member having dynamically allocated memory**) – provide deep copy instead of shallow copy.**

Copy Constructor

Example

```
class Person {
    private :
        char * name;
        int age;
    public :
        Person(); //default
        Person(char *, int);
        Person(const Person &); // object is passed by reference
        ~Person();
        void Read();
        void Write() const;
}
```

Copy Constructor

Example Contd...

```
Person::Person(const Person & rhs): age(rhs.age){
    name = new char[strlen(rhs.name)+1]),
    strcpy(name, rhs.name);
}

int main() {
    Person p("raman", 53);
    p.write();

    Person q(p); // q is a new object instantiated by an existing
object p using copy constructor
    q.write();

    Person r = p; // r is a new object instantiated by an existing
object p using copy constructor
    r.write();

}
```

Assignment Operator

There are instances in the code where an object is initialize with an existing object using assignment operator.

Example code :

```
int main() {
    Person p("raman", 53);
    p.write();
    Person q; // q is a new object instantiated
    q = p; // q object updated by an existing object p using
  assignment operator
    q.write();
}
```

A default implementation of assignment operator is provided by the compiler that does a member wise copy (**shallow copy**). This will not work if the object has a pointer.

In that case programmer is required to provide an implementation of assignment operator if the object has some resource (for example : pointer data member having dynamically allocated memory**) – provide deep copy instead of shallow copy.**

Assignment Operator

Example

```
class Person {
    private :
        char * name;
        int age;
    public :
        Person(); //default
        Person(char *, int);
        Person(const Person &);
        Person & operator=(const Person &);
        ~Person();
        void Read();
        void Write() const;
};
```

Assignment Operator

Example Contd...

```
Person & Person::operator=(const Person &rhs) {
    name = new char[strlen(rhs.name)+1];
    strcpy(name, rhs.name);
    age = rhs.age;
    return *this;
}
int main() {
    Person p("raman", 53);
    p.write();

    Person q; // q is a new object instantiated
    q = p; // q object updated by an existing object p using
assignment operator
    q.write();

    p = p; // Also valid operation, but has side effect
    Person r("ravi", 33);
    r = p; // Also valid operation, but has side effect
}
```

Assignment Operator

Enhanced code

```
int main() {
    Person p(“raman”, 53);
    p = p; // This is valid operation, but may cause memory leak

    Person r(“ravi”, 33);
    r = p; // This is valid operation, but may cause memory leak
}

Person & Person::operator=(const Person &rhs) {
    if(this == &rhs) // self checking
        return *this;
    delete []name; // remove whatever existed before

    name = new char[strlen(rhs.name)+1];
    strcpy(name, rhs.name);
    age = rhs.age;
    return *this;
}
```

Friend Function

There are instance where a function might be required to access the data members of a class even though they are private and this function can not be made member of a class.

In such case, we declare that this function is a **friend** of the class. A friend function can access the private members of the class to which it is a friend.

Note that a friend function can be member of one class also.

Friend Function

Example code

```
Example code :


class Person {
    private :
    char * name; int age;
    public :
    ...
    friend void doctor(Person &);
    // friend declaration
generally  placed in public
section
};
```

```
void doctor(Person & p) {
    cout << "Administrating age
reduction tonic" << endl;
    p.age = p.age - 10;
    // accessing private member
}


int main() {
    Person p("suparman", 50);
    p.write();
    doctor(p);
    p.write();
    return 0;
}
```

Friend Class

There are instances where a class might require to use another class – the former class may want to access private members of the latter class. The former class is made friend to the latter class.

```
Example code : Please
complete by your own


class Node {
    private :
        int info;
        Node* link;
        Node(int) //
private constructor
        friend class Queue;
};
```

```
class Queue {
    public :
        Queue();
        ~Queue();
        void add(int); // Add at rear
        void remove(); // Remove from
front
        bool isEmpty();

    private :
        Node * f, *r;
        // front and rear pointers
};
```

Operator Overloading

Operator Overloading for Assignment Operator (=) already discussed

For your detail study !!

Static Members

A Class may have static data members and static member function

- A static data member belongs to the class and not to each object – any information related to the class can be maintained in the static data member
- All the objects of the same class will share the same static data member
- A static member is declared in the class definition like any other non-static data member, but it has to be <u>defined</u> in one of the implementation files explicitly
- A static member function can only access the static data members
- A non-static member function can also access the static data members of the class

Static Members

Example

```
class Person {
    private :
        char * name;
        int age;
        static int personCount;
    public :
        Person() { personCount++ } // Constructor accesses the
static data member
        ...
        ~Person() { personCount-- } // Destructor accesses the
static data member
        static void DisplayNumOfPerson();
};

int  Person::personCount = 0; // definition of the static data
member
```

Static Members

Example contd...

```
// definition of the static member function
void Person::DisplayNumOfPerson() {
    cout << "Number of Person : " << personCount << endl;
}
int main() {
    Person::DisplayNumOfPerson(); // displays 0
    Person a;
    a.DisplayNumOfPerson(); // displays 1
    {
        Person b;
        b.DisplayNumOfPerson(); // displays 2
        a.DisplayNumOfPerson(); // displays 2
    }
    Person::DisplayNumOfPerson(); // displays 1
}
```

Class Template

There are many data structures which have certain properties independent of the components they contain. For example :

- **Queue** : It has property of FIFO, doesn't not matter what type of component we put in the queue
- **Stack** : It has property of LIFO, doesn't not matter what type of component we put in the stack

In c++ we can write such generic data structure using **template class** concept.

In this approach, the class as well as the member functions are generated at compile time based on the definition of the object in the client code.

Like -

```
MyStack<int> s1(10); // stack of 10 integer
MyStack<double> s2(5); // stack of 10 integer
```

To carry out this process, the code has to be exposed to the client.

Class template

Example

```
template <class T> // can use the keyword typename also
class MyStack {
    public :
        MyStack(int m = 10);
        void Push(T x);
        T Pop();
        ...
    Private :
        T * elementList;
        ...
};
int main() {
    MyStack<int> S1(10);    S1.push(5);
    MyStack<double> S1(5);  S2.push(9.33);
}
```