TM
An ISO 9001:2008 Certified Company

numeric
infosystem private limited

C++ PROGRAMMING for the absolute beginner

C++ PROGRAMMING

"FIRST SOLVE THE PROBLEM. THEN, WRITE THE CODE "

Syllabus - Campus Recruitment Training (CRT)

Description of Training on Quantitative Aptitude and Reasoning

- Number System
- Logarithms
- Average
- Problem on Ages
- Percentages
- Profit & Loss
- Ratio & Proportion
- Time & Work
- Pipes & Cisterns
- Time, Speed & Distance
- Allegations and Mixture
- Simple Interest & Compound Interest
- Permutation & Combination
- Probability
- Geometry

LOGICAL REASONING

- Analogy /Series Completion
- Coding – Decoding
- Blood Relations
- Puzzle test
- Sitting Arrangement
- Direction Sense Test
- Logical Venn Diagrams
- Number, Ranking & Time Sequence Test
- Data Sufficiency
- Crypt arithmetic Questions

10 Campus Drive For Registered Students

VERBAL ABILITY

- Reading Comprehension
- Jumbled Paragraph Questions
- Vocabulary Based Questions
- Fill in Blanks
- Miscellaneous Questions
- Error identification
- Error correction
- Antonyms
- Synonyms / Ordering sentences

TECHNICAL INTERVIEW PREPARATION

Topics to be covered in C

- Basic Concepts of C Language.
- Basic Programming Skills.
- Arrays, Pointers & Structures.

Topics to be covered in C++:

- Introduction to C++
- Object-Oriented Programming Concepts
- The Basics of C++
- Pointers and Arrays
- Function and Operator Overloading
- Reusing classes
- Virtual functions and Polymorphism

Topics to be covered in Datastructures:

- Introduction to datastructures.
- Stacks/Queue
- Linked list
- Tree
- Sorting Algorithms.

Topics to be covered in Databases:

- Introduction to Database.
- Introduction to Normalization.
- Introduction to Data Definition Language/DML.
- Introduction to SQL/SQL Functions.
- Introduction to Set Operators, Groups, Reports.

SOFT SKILLS

- Training need analysis (TNA)
- Behavioural training
- English language & Communication training
- Group Discussion
- Resume \ Curriculum vitae
- Interview Skills
- Email Etiquettes
- Business Etiquette and Customer Handling
- Etiquette of dressing
- Mock Group Discussions
- Mock Interviews
- Mock Recruitment Drive

Course Fee: ₹ 9999/-

Special Features:-

1. Comprehensive Study Material/ Practice Sheets/ Test paper.
2. Online Mock test will be conduct after the completion of every topic.
3. 8 Diagnostic Career Test and 2 Pre-Assessment Test
4. Get directly interview calls from companies only after a good score in Pre-Assess.

PART ONE OOPS WITH C++

Since the invention of computers so many programming approaches and technique have been tried. These techniques includes such as:

- ✓ **Top to bottom approach**
- ✓ **Modular programming**
- ✓ **Structured programming**
- ✓ **Bottom to top approach**

The primary motivation in each case is to eliminate the complexity of program and make language more reliable and portable.

C is a structured programming language which becomes popular in 1980’s. Structured programming was a powerful tool which enable a user to write complex program and modules fairly easy. However as a programs grew larger, even the structured approach failed to show the desired output in terms of bug *free*, easy to maintain and reusable program.

Object Oriented Programming (OOP) is an approach which eliminate some pitfalls of conventional programming methods and incorporate some new technique and several powerful concept in structured programming and thus make language more reusable, maintainable and efficient.

Procedure Oriented Language.

This is the most popular tech. Of 1980’s. POP employs top down programming approach where a problem is viewed as a sequence of tasks to be performed. A number of functions or procedures are written to attempt these tasks, Means this language is totally function/procedural oriented.

Some characteristics of POP are:

- ✓ Emphasis is on doing things (Algorithms)
- ✓ Large programs are divided into small programs known as functions.
- ✓ Most of the function share global data
- ✓ Data move openly around the system from function to function
- ✓ Functions transform the data from one form to another

Employs top-down approach in program design.

Drawbacks of POP

1) Data move openly around the system and are there for vulnerable changes caused by any function in the program.
2) As a programs grew larger, even the structured approach failed to show the desired output in terms of bug free, easy to maintain and reusable program.
3) It does not model very well the real world problems.

Object Oriented Programming (OOP).

Object Oriented Programming (OOP) was invented to overcome the drawbacks of POP.

Every thing in this world is an Object and the human being in this real world is very close to these objects because these objects are used by human beings. So if programming tech. Is modeled as object oriented it is very easily to handled and understand by the programmers or users. Thus this programming tech. is also kwon as real world/real time programming.

Oops treated the Data as a critical element and dose not allow it to flow freely in a program, it binds the data more closely to the function which operate on it and protect it from accidental modification in a data structure called class. This feature is called encapsulation.

Some feature of OOPS are

- ✓ Emphasis is on data rather then procedure.
- ✓ Programs are divided into what are known as object
- ✓ Data structure are design in such that they known as object.
- ✓ Function that operate on the data of an object are tied together in a data structure
- ✓ Data is hidden and cannot access by any external function even main
- ✓ Objects may communicate with each other through function
- ✓ New data and functions are easily insert when require
- ✓ Follow bottom to top approach

Components of OOPS

- ✓ ***Classes:***

Classes can be defined using keyword class. A class declaration is used to create a new identifier kwon as object. The entire set of data and code of an object can be made a user define type with the help of a class. Object are variable of the type class. Once class has been define, we can create any number of objects belonging to that class. A class is thus a collection of objects of similar types.

Note: 1) Classes are Logical Abstraction While Objects Have Physical Existence.

2) Objects are Instance of class.

To prepare a Class in C++ one can follow following Syntax:

```
Class <class name>
{
  private:
         member functions and data
  public:
        member functions and data
};
```

A class declaration is similar to struct declaration. The keyword class specifies, that what follows is an abstract data of type <class name>. A class generally contains two types of member.

1) Private

2) Public

Members which declare private within the class can be accessed only within the class. On other hand public members can be accessed from outside the class also.

Private members: This section is used to define abstract or hidden member. Members which are declared in this section are totally hidden and can't be accessed by any external function even main(). Only the member function which are declared within the class can access these members.

Public members: Members which are declared   in this section can be accessed by any external function even main with the help of an Objects.

- ✓ **Objects:**

Objects are instance of class. These are basic run time entities of object oriented programming system. They may represent person, place, bank account etc.

Object may defined by

Identity, state, behaviour

Any Objects can be Identified by their behaviour and this particular behaviour assigns them a unique name. State represent the nature of object.

Some properties of objects

- ✓ Objects are real world entities which contains data and process both.
- ✓ Programming problem is analyzed in terms of objects.
- ✓ Objects take memory space in the memory and have associated address.
- ✓ When a program is executed, the objects interact by sending message to one another.
- ✓ For example, if “customer” and “account” are two objects in a program, then the customer object may send a message to the account object requesting for bank balance.

Creating Objects:

If Fruit is a class then Objects are Orange, Apple etc;

Ex:

Fruit Orange, Apple, Mango;

- ✓ **Data Abstraction and Encapsulation**

The wrapping up of data and function within a single unit(class) is known as encapsulation. Encapsulation is the way to implement the data abstraction.

Data encapsulation is the most striking feature if the class. The data is not accessible to the outside world, only those function which wrapped in the class can access it. These function are interface between object data and the program. This insulation of the data from the direct access by the program is called Data Hiding.

Abstraction refers to the act of representing essential features without including the background details of explanations. Classes uses the concept of data hiding means the member which declare in private section of class can't access by any external function.

This feature is also known as ADT(abstract data type).

Ex. To understand the abstraction lets take an example of car, you anly know the essential feature of car like gear handling , steering handling, use of break and clutch etc.

But while driving a car you don't know about the engine, gear, break, clutch, how it is

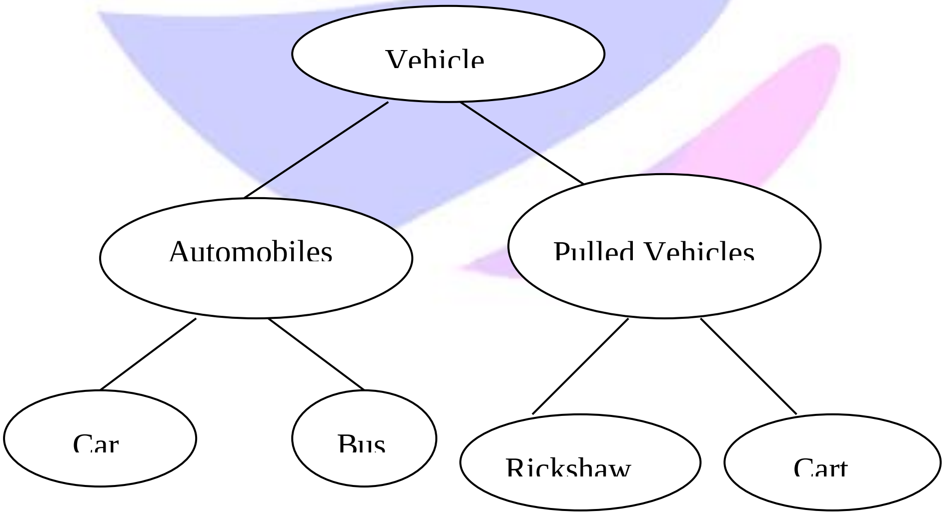
Internally working.

- ✓ **Inheritance:**

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

Role of Inheritance in oops

1. Its capability to express the inheritance relationship which makes it ensure the closeness with the real world models.
2. Another is the idea of reusability. Inheritance allows the addition of additional features to an existing class without modifying it.
3. The third reason is its transitive nature. If a class A inherits properties of another class B, then all subclass of A will automatically inherit the properties of B.



The class whose properties are inherited, is called Base Class or Super Class and the class that inherits these properties, is called Derived class or sub class.

Vehicles is the base class of automobiles and pulled vehicles, means both access the properties of Vehicles. Car and bus are the sub class of automobiles. automobiles is the base class of Car and Bus.

- ✓ **Polymorphism:**

Polymorphism is the another feature OOPS . Polymorphism is a Greek term, means the ability to take more than one form. Polymorphism is also introduces reusability, reuse of same name or operation for a different different purpose.

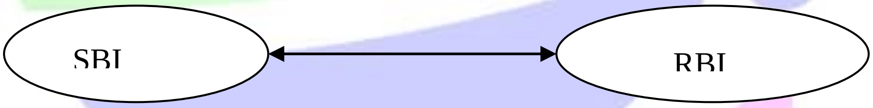
For Example if '+' operator is used to add two numbers ,its also used to add two strings means a '+' symbol by default is used to add two numbers but when it is reuse to add two strings then its behaviours gets change and it introduces the concept of polymorphism .

Ex: A process DRAW is used to draw a rectangle, triangle ,circle etc .So here DRAW is process which is reused for three purpose.

- ✓ **Message Passing**:

This technique is used to Establish Communication between Two or more Objects.

Ex. If two Banks(two objects) wants to establish communication the Message passing is used.



StaticMember Data.

A data member of a class can be qualified as static. the properties of a static member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are:

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects

Notice the following statement in the program:

Int item :: count; // definition of static data member

Note that the type and scope of each static member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as class variables.

Static Member Function

Like static member variable, we can also have static member functions. A member function that is declared static has the following properties:

- A static function can have access to only other static members (functions or variables) declared in the same class.
- A static member function can be called using the class name (instead of its objects) as follows:

A Static Member Function can be called in following Manner.

Class-name :: function-name;

Friendly functions

The function declaration should be preceded by the keyword **friend** the function is defined elsewhere in the program like a normal C++ function A function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access rights to the private member of the class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as **friend** .
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member function, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.(e. g .A. x)
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

Syntax

```
friend  <return type><fn name>(arguments)
{

===
}
```

Inline Function

✓ One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

✓ To eliminate the cost of calls to small function, C+ + propose a new feature called *inline function*. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code . The inline function are defined as follows:

```
Inline  function-header
{
       function
}
  Example
 inline double cube (double a)
{
       return (a*a*a);
}
```

Some of the situations where inline expansion may not work are:

1) For functions returning values, if a loop ,a **switch,** or a **goto** exists.
2) For functions not returning values, if a return statement exists.
3) If functions contain **static** variables.
4) If **inline** functions are recursive.

Note: Inline expansion makes a program run faster because the overhead of a function Call and return is eliminated. However, it makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the

function is called.

Default Argument.

- C+ + allows us to call a function without specifying all its arguments . In such cases, the function assigns a *default value* to the parameter which does not have a matching argument in the function call.
- Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values.
- Here is an example of a prototype (i.e. function declaration) with default values:

  float amount (float principal , int period , float rate=0.15);

  The above prototype declares a default value of 0.15 to the argument **rate,**

A default argument is checked for type at the time of declaration and evaluated at the time of call. *One important point to note is that only the trailing arguments can have default values. It is important to note that we must add defaults from right to left.* *We cannot provide a default value to a particular argument in the middle of an argument list .some examples of function declaration with default values:*

```
int mul   (int i, int  j=5, int   k=10);              // legal
int mul   (int  I=5,  int j);                         // illegal
Int mul   (int  i=0,  int j,  int  k=10);             // illegal
Int mul   (int  i=2,  int j=5, int k=10);             // legal
```

Advantage of Default Arguments:

- Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit.
- It also provides a greater flexibility to the programmers., .A function can be written with more parameters than are required for its most common application. Using default arguments, a programmer can use only those arguments that are meaningful to a particular situation.

Function OverLoading:

- ✓ As stated earlier, overloading refers to the use of same thing for different purpose. C++ Also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in OOP.
- ✓ Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists, The function would perform different operations depending on the argument list in the function call.
- ✓ The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded **add ()** function handles different types of data as shown below:

```
// Declarations

int  add (int   a,   int    b);                        // prototype 1

int  add ( int  a, int     b, int    c );              // prototype 2

double   add (double x,   double  y);                  // prototype 3

double   add (int   p,  double q);                     // prototype 4

double   add (double p, int q);                        // prototype 5

// function calls

cout << add (5,  10);                           // uses  prototype 1

cout <<  add (15, 10.0 );                       // uses  prototype 4

cout << add (12.5,7.5);                        // uses  prototype 3

cout <<add (5,  10,15);                        // uses  prototype 2

cout <<add (0. 75, 5 );                        // uses  prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match be unique.

The function selection involves the following steps:

- ✓ The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
- ✓ If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,
  - **Char** to **int**
  - **Float** to **double**

  To find a match.
- ✓ When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique.
- ✓ If all of the step fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Constructor

- ✓ A constructor is a `special` member function whose task is to instantiate as well as initialize the objects of its class. It is special because its name is the same as the class name.
- ✓ The constructor Is invoked whenever an object of its associated class is create.
- ✓ It is called constructor because it constructs the values of data members of the class.

```
 // class with a constructor
   class integer
  {
         int   m,    n;
      public:
          integer (void );    // constructor declared
          …..
          …..
};
integer  ::  integer  (void)       //constructor   defined
```

```
{    m= 0; n = 0;
}
```

when a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```
integer   intl;        //object intl created
```

not only creates the object **intl** of type **integer** but also initializes its data members **m** and **n** to zero. There is no need to write any statement to invoke the constructor function (as we do with the normal member function) If a `normal ` member function is defined for zero initialization, we would need to invoke this function for each of the objects separately , this would be very inconvenient, if there are a number of objects .

A constructor that accepts no parameters is called the default constructor. The default constructor for ***class A is A::A() .****if no such constructor is defined, then the compiles supplies a default constructor.*

The constructor function have some special characteristics. These are:

- They should be declared in the public section
- They are invoked automatically when the objects are created .
- They do not have return types, not even void and therefore, and they cannot return values.
- 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.

Copy Constructor

As stated earlier, a copy constructor is used to declare and initialize an object from another object. For example, the statement

Integer A (B);

Would define the object A and at the same time initialize it to the values of B. Another form of this statement is

Integer A = B

The process of initializing through a copy constructor is known as copy initialization. Remember, the statement

A = B;

will not invoke the copy constructor . However, if A and B are objects, this statement is legal and simply assigns the values of B to A.

Properties of copy constructor.

- ✓ *A copy constructor takes a reference to an object of the same class as itself as an argument.*
- ✓ *A reference variable has been used as an argument to the copy constructor . we cannot pass the copy argument by value to a copy constructor.*
- ✓ *When no copy constructor is defined, the compiler suppliers its own copy constructor .*

Destructor.

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 . for example, the class integer can be defined as shown below:

~ integer () { }

Properties of Destructors

- ✓ 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 (or block or function as the case may be) 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 .

- ✓ Whenever new is used to allocate memory in the constructors, we should use delete to free that memory . for example, the destructor for the matrix class discussed above may be defined as follows:

```
   Matrix   ::  ~matrix ()
{
for (int I =0 ; i<d1;   i++)
delete p[i];
delete p;
}
```

this is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

Operator Overloading

- ✓ The Operator overloading is one of the many exciting features of C++ language . It is important technique that has enhanced the power of extensibility of C++
- ✓ This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.
- ✓ Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can almost create a new language of our own by the creative use of the function and operator overloading techniques. We can overload (give additional meaning to) all the C++ Operator Except the following:

- Class member access operators (., .*).
- Scope resolution operator (::).
- Size operator **(sizeof)**
- Conditional operator (?:)

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of operator function, called operator function, which describes the task. The general form of an operator function is :

```
<return type>   class name   ::   operator <op>(arglist)
    {
        function body     // task defined
    }
```

where< return type >is the type of value returned by the specified operation and <op> is the operator being overloaded. the<op> is preceded by the keyword **operator . operator** op is the function name .

Operator functions must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will only one argument for unary operators and two for binary operators, while a member function has no argument for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions.

OOPS – PART – TWO

Inheritance

Reusability is yet another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to cerate the same all over again. It would not only save time and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones the mechanism of deriving a new class from an old one is called inheritance (or derivation). The old class is referred to as the base class and the new one is called the derived class or subclass.

The derived class inherits some or all of the traits form the base. A class can also inherit properties from more than one class or from than one level. A derived class with only one base class, is called single inheritance and one with several base classes is called multiple inheritance . on the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another derived class is known as multilevel inheritance.

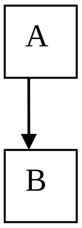
```
Class  derived-class-name     : <Vsibility –mode>  Base-class –name
{
    …….//
    ……..//   members of derived class
    …….//
};
```

The colon indicates that the derived-class-name is derived from the base-class-name. The visibility-mode is optional and, if present, may be either **private** or **public.** The default visibility –mode is **private.** Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

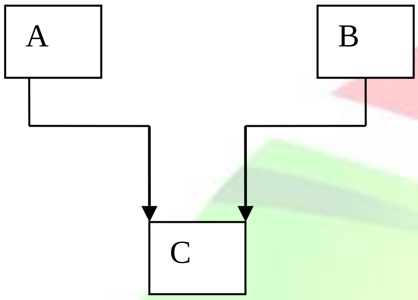
When a base class is privately inherited by a derived class, `public members` of the base class become `private members` of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class they are inaccessible to the objects of the derived class. Remember, a public member of a class can be accessed by its own objects using the dot operator . The result is that no member of the base class is accessible to the objects of the derived class .

On the other hand, when the base class is publicly inherited, `public members` of the base class become `public members` of the derived class and therefore they are accessible to the objects of the derived class. In both the cases, the private members are not inherited and therefore, the private members of a base class will never become the members of its derived class.

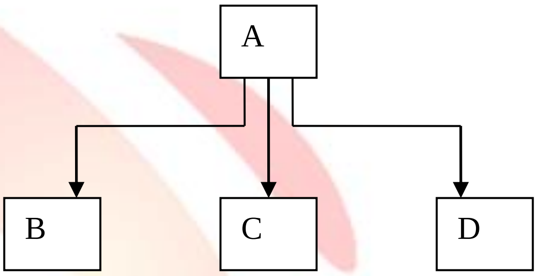
Types of Inheritance:



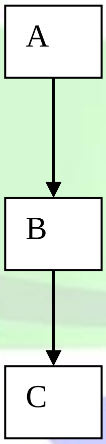
(a) Single inheritance



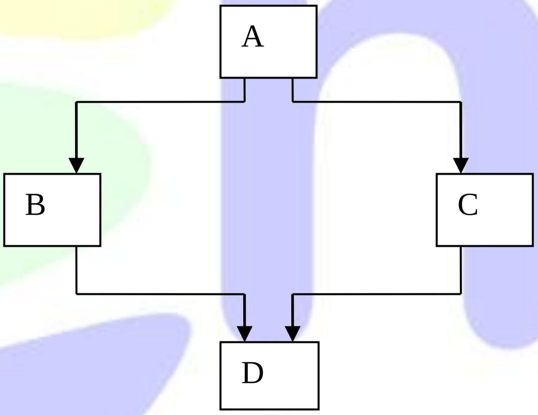
(b) Multiple inheritance



(c) hierarchical inheritance



(d) Multilevel inheritance



(e) Hybrid inheritance

Protected Members

We have just seen how to increase the capabilities of an existing class without modifying it. We have also seen that a private member of a base class cannot be inherited and therefore it is not available for the derived class directly. What do we do if the private data needs to be inherited by a derived class? This can be accomplished by modifying the visibility limit of the private member by making it public. This would make it accessible to all the other functions of the program, thus taking away the advantage of data hiding.

C++ provides a third visibility modifier, **protected,** which serve a limited purpose in inheritance. A member declared as protected is accessibly by the member functions within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes. A class can now use all the three visibility modes as illustrated below:

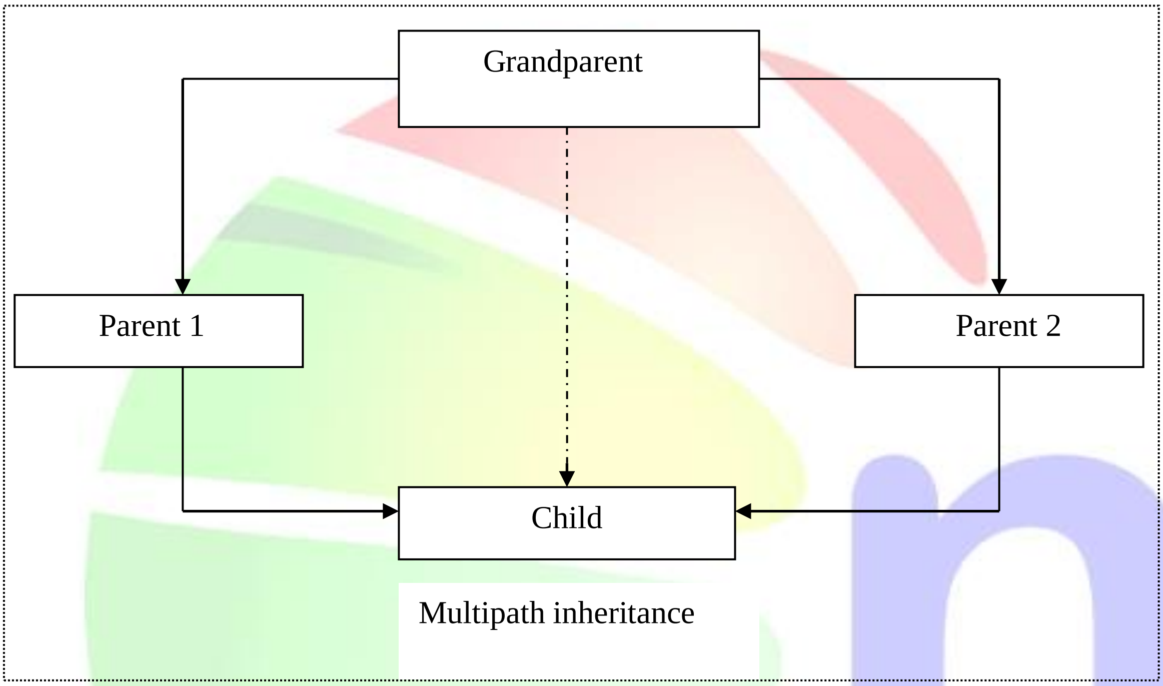
```
Class alpha
{
        private :               //  optional
          …….                   //  visible to member functions
          …….                 // within its class

        protected:
          …….             // visible to member functions
          …….             // of its own and derived class
        public:
           ……             // visible to all functions
           ……             // in the program
 }
```

When a protected member is inherited in public mode , it becomes protected in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A protected member, inherited in the private mode derivation, becomes private the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since private members cannot be inherited).

Virtual Base Class

Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in the figure that `child` has two direct base classes `parent1` and `parent2` which themselves have a common base class `grandparent`. The `child` inherits the traits of `grand parent` via two separate paths. It can also inherit directly as shown by the broken line. The `grandparent` is sometimes referred to as indirect base class.



Multipath inheritance

Inheritance by the `child` as shown in might pose some problems. All the public and protected members of `grandparent` are inherited into `child` twice, first via `parent1` and again via `parent2` this means, `child` would have duplicate sets of the members inherited from `grandparent`. This introduces ambiguity and should be avoided.

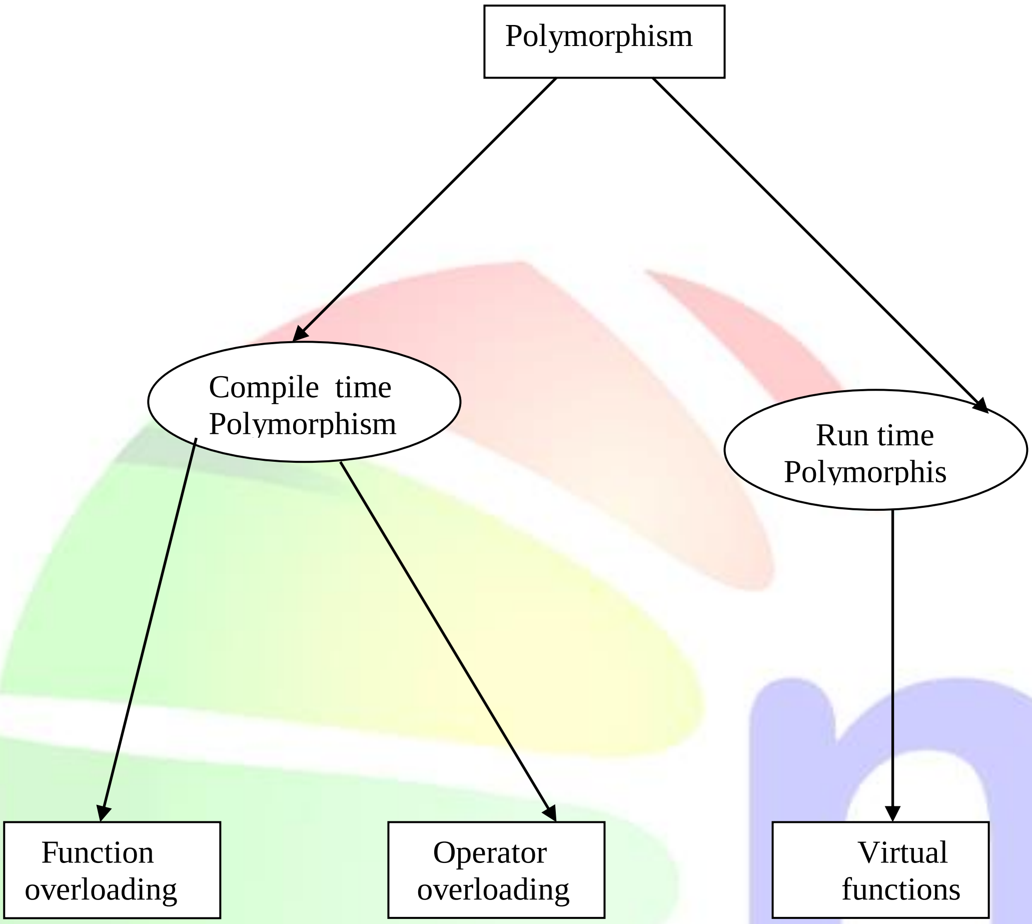
The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class)as virtual base class while declaring the direct or intermediate base classes which is shown as follow:

```
Class A                         // grandparent
 {
   ……
   ……
};
class B1 : virtual public A        // parent1
{
   ……
   ……
};
class B2 : public virtua1 A        // parent 2
{
   ……
};
class  C :public  B1,  public B2 // child
{
   ……             //only one copy  of A
   ……             // will be inherited
};
```

When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.

Note: The keywords virtual and public may be used in either order.

Polymorphism



Achieving polymorphism

Polymorphism is one of the crucial features of OOP. It simply means `one name , multiple forms. We have already seen how the concept of polymorphism is implemented using the overloaded functions and operators. the overloaded member functions are `selected` for invoking by matching arguments, both type and number. This known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early binding or static binding or static linking . Also known as compile time polymorphism, early binding simply means that an object is bound to its function call compile time.

At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as late binding. It is also known as dynamic binding.

Virtual Functions(Redefining Methods)[Late Binding, Run Time Polymorphism]

Polymorphism refers to the property by which objects belonging to different classes are able to the same message, but in different forms. As essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes. This required the use of a single pointer variable to refer to the objects of different classes. ***Here, we use the pointer to base class to refer to all the derived objects. But, we just discovered that a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. How do we then achieve polymorphism ? It is achieved using what is known as `virtual` functions.***

When we use the same function name in both the base and derived classes, the function in base class is declared as virtual the using the keyword virtual preceding its normal declaration . when a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of pointer. Thus, by making the base pointer to point to different object, we can execute different versions of the virtual function.

<u>When virtual functions are created for implementing late binding., we should observe some basic rules that satisfy the complier requirements :</u>

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers .
4. A virtual function can be a friend of anther class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If tow functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored .

7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say , we cannot use a pointer to a derived class to access an object of the base type.
9. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Pure Virtual Function [Abstract Method/Abstract Class]

It is normal practice to declare a function virtual inside the base class and redefine it in

the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder. For example, we have not defined any object of class media and therefore the function display () in the base class has defined `empty` such functions are called “do-nothing” functions.

A “do-nothing” function may be defined as follows :

Virtual void display () = 0;

Such functions are called pure virtual functions. pure virtual function is a function declared in base class that has no definition relative to the base class. In such cases, the complier requires each derived class to either define the function or redeclare it as pure virtual function. Remember that class containing pure virtual functions cannot be used to declare any objects of its own .As stated earlier, such classes are called abstract base classes and to create a base pointer required for achieving run time polymorphism.

this Pointer

The **this** pointer is used as a pointer to the class object instance by the member function. The address of the class instance is passed as an implicit parameter to the member functions. C++ keeps only one copy of each member function and the data members are allocated memory for all of their instances. This kind of various instances of data are maintained use **this** pointer.

Important notes on this pointer:

- **this** pointer stores the address of the class instance, to enable pointer access of the members to the member functions of the class.
- **this** pointer is not counted for calculating the size of the object.
- **this** pointers are not accessible for static member functions.
- **this** pointers are not modifiable.

OOPS PART-3

Abstraction

- ✓ **Abstract classes**

A class that contains one or more abstract methods and therefore can never be instantiated. Abstract class are defined to the other classes can extend them and make them concrete by implementing the abstract methods. An abstract class has no direct instances but whose descendent classes have direct instances. Its objects cannot be created and the subclass of abstract class may have instance(object). The function defined in abstract class does not have any body. It is used as a foundation for derived classes.

Abstract classes are defined just as ordinary classes. However, some of their methods are designated to be necessarily defined by subclasses. We just mention their *signature* including their return type, name and parameters but not a definition. One could say, we omit the method body or, in other words, specify “nothing ‘’.this is expressed by appending “=0” after the method signatures.

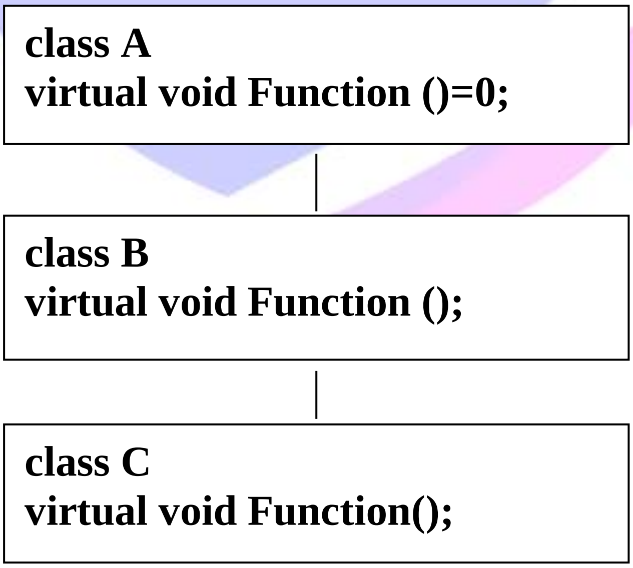
An abstract class contains at least one pure virtual function. Object of abstract class cannot be created.

Example:

```
class Media
{
public:
virtual void Show ()=0;
};
void main ()
{ Media *A
A= new A;  //cannot create object of an abstract class
A->Show();}
```

This class definition would force every derived class from which objects should be created to define a method Show (). These method declarations are also called pure methods.

Abstract classes are set up for the benefit of derived classes. Abstract normally occur at the root of the hierarchy .there can be any number of abstract classes in the same hierarchy.



```
class A
virtual void Function ()=0;

class B
virtual void Function ();

class C
virtual void Function();
```

An abstract class with one or more pure virtual functions has the following properties:

1. Describe an unrealized concept.
2. Objects of an abstract class type cannot be created.
3. Derived classes can be built from these abstract classes.
4. Objects of the derived classes can be created.

See Virtual Function

- ✓ **Abstract Method:**

When derive class has same signature of method as in base class then a complexity occurred called Method overriding. Method Overriding is basically implemented in inheritance to extend the functionality of Member Function. The main task of Method Overriding is to apply run time polymorphism (RTP).

To Apply RTP, We have to define abstract method(do nothing method) within the base class. Abstract Method have following properties.

1) A method which have no body structure is known as abstract method(this type of method define in c++ by assign value Zero to method Ex: void Show()=0;
2) A Class that contains abstract method is called abstract class, the main property of abstract class is that its object can't be instantiated.
3) Once abstract method define in base class it must be redefine or implemented in derive class other wise compile time error occurred.
4) The main advantage of abstract method is that one can call or activate variety of Method (redefine method) with different behavior at run time.

See Virtual Function

- ✓ **Data Abstraction and Encapsulation**

The wrapping up of data and function within a single unit(class) is known as encapsulation. Encapsulation is the way to implement the data abstraction.

Data encapsulation is the most striking feature if the class. The data is not accessible to the outside world, only those function which wrapped in the class can access it. These function are interface between object data and the program. This insulation of the data from the direct access by the program is called Data Hiding.

Abstraction refers to the act of representing essential features without including the background details of explanations. Classes uses the concept of data hiding means the member which declare in private section of class can't access by any external function.

This feature is also kwon as ADT(abstract data type).

Ex. To understand the abstraction lets take an example of car, you anly know the essential feature of car like gear handling , steering handling, use of break and clutch etc.

But while driving a car you don't know about the engine, gear, break, clutch, how it is Internally working.

- ✓ **Early binding and late binding**

Early Binding: in most traditional programming languages such as C and

PASCAL, the compiler call fixed function identifiers, based on the source code. In early binding, the function identifiers are associated with physical addresses before runtime and during the process of compilation and linkage. The compiler calls fixed function identifiers, based on source code. The linker takes these identifiers and replaces them with a physical address. Examples:

1. Standard function calls.
2. Overloaded function and operator calls.

Advantage: it is faster and often requires less memory.

Disadvantage: lack of flexibility.

Late binding: A function call is only indicated in the source code, without specifying the exact function to call is known as late binding. In nutshell, function calls are resolved at run time.

Advantages:

1. Allows greater flexibility.
2. Used to support a common interface, while allowing various objects utilizing this interface to define their own implementations.
3. Used to create class libraries that can be reused and extended. Disadvantages: little loss of execution speed.

See Virtual Function

✓ Association

Association describes a group of links with common structure and common semantics. All links in an association connect objects from the same class .it describes a set of potential links. Associations are bi-directional. The implementation of associations through programming languages is as pointer from one object to another. Pointer is an attribute of an object that contains an explicit reference to another object.

Associations are important because they define the routes for message passing between objects.

Importance of associations

1. Association has been widely used throughout the database modeling.
2. Most object-oriented language implement associations with object pointers.
3. Associations violate encapsulation of information into classes.

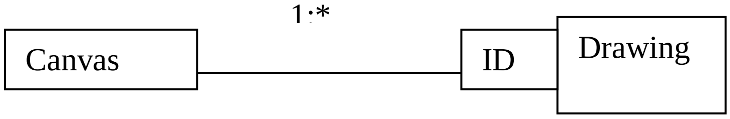
Associations are a link between objects of two or more classes. An association exists between objects of different classes whenever there is a logical linking, interaction, or dependency between the objects. An association relationship may be described as “has a”, “associated with” or “knows about”.

An association has a cardinality or multiplicity. This is the number of objects on one side of an association that may be associated with an object on the other side of the association. For example, one car has one passenger. This is an example of a l to l association. If a car object may have many passenger objects, then there is a l to many association between the car class and the passenger class.

The association relationship is important for many reasons. Association is the most fundamental relationship of real world physical things and software entities. Association

helps develop highly cohesive objects where all associated objects are linked to accomplish a purpose. It implements dependency among objects. For example an object may be dependent upon its associated objects to accomplish its function. It implements visibility among objects for example, an object has visibility for (sees) its associated objects so that it can send messages to its associated objects.

Identifying Associations (Qualified Associations)



Usually a name identifies an Object within some context, this name uniquely identify an object. Drawing class has Object (ID) or qualifier like Circle, Square, and Cube etc.

A Qualifier

Distinguishes objects on the “many” side of an association.

Identify association by looking for logical links or connections between two or more objects. One object may “know about” another object. For example, a retail store “knows about” its employees. One object may “depend upon “another object. For example, a bank “depends upon” its tellers. Often it is useful to make a drawing of the key objects in a system and show the association between the objects on the drawing. For each object on the drawing, identify associations between them.

The following are suggested steps to identify association relationships:

1. Make a drawing or physical representation, e.g. a car, a motor a passenger.
2. Identify the objects and their classes, e.g., a car, a motor, a passenger.
3. for each object, identify association relationships with the question “for each object what are the associated objects ?”
4. Identify the multiplicity of each association relationship with the questions “this object is associated with zero, one or many of the other object ? and “this association is either optional or required?
5. Identify if the relationship is an association (“has”) or an aggregation (“part of “) when in doubt assume association (“has”).
6. For each relationship, identify the constraints (limitations or rules).
7. If available, check messages between objects in the dynamic model because messages in dictate association and aggregation relationship.
8. Create a class diagram, data dictionary, class specification, and prototype.

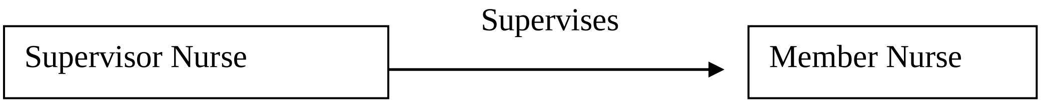
Recursive Association:

A recursive association is where a class is associated to itself. In the recursive association only one class is involved this is the type of association used with linked lists.

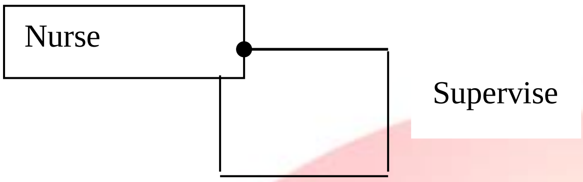
Example of Recursive Association

The relationship of supervisor nurse supervises nurse(member) can be expressed in two different model.

Model A:



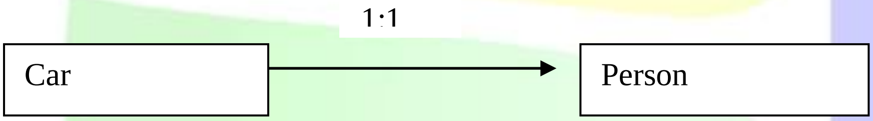
Model B:



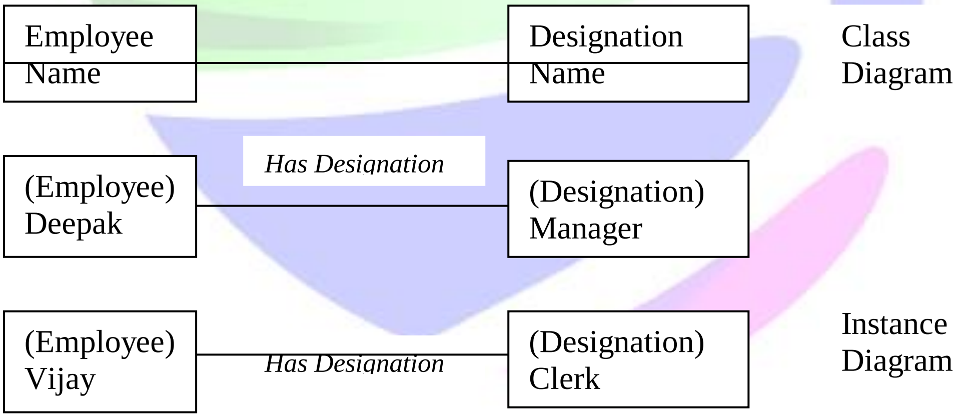
Consider: is there any different behavior between supervisor nurse and member nurse? If no, then all we need to do is to show the relationship between them and a single class with a recursive association (model B) is sufficient. However, if the answer is yes, then we need two separate classes (model A) to handle the different requirements (like the example of Supervisor Nurse Supervise Member Nurse –two classes are needed to set the Relationship).

One to One Association

One to one association shows a very narrow association.



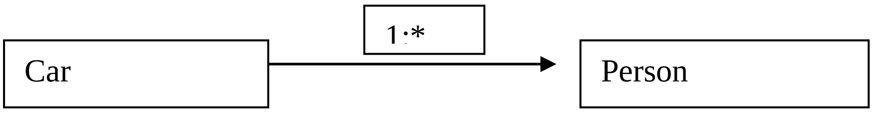
The above model shows the one to one association between objects that a one car is used by one person



Each association in the class diagram corresponds to a set of links in the instance diagram, just as each class corresponds to a set of objects. Each employee has Designation. Designation is the name of the association.

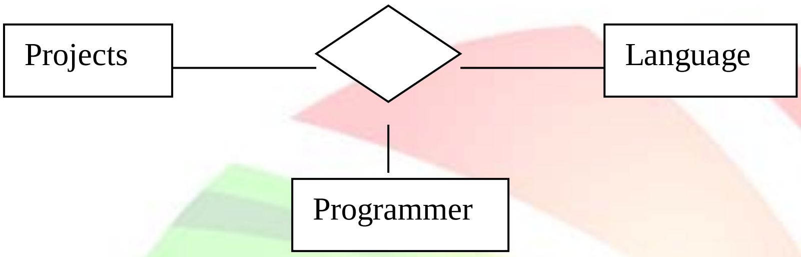
Many to many association

Many associations are traversed in both directions, although not usually with equal frequency, his approach permits fast access, but if either attribute is updated then the other attribute must also be updated to keep the link consistent there are many approaches to their implementation:

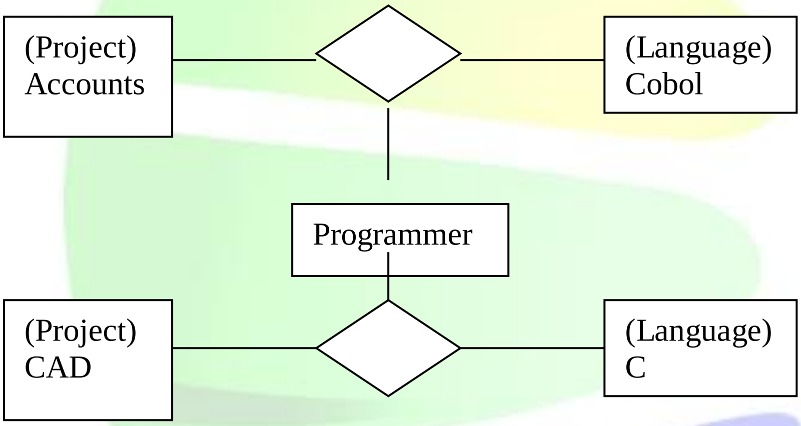


The above model shows the one to many association between objects that a one car is used by many person.

Ternary Association: The Diamond Shape symbol is used to define Ternary Association between objects.



Class Diagram



Instance Diagram

In above model a programmer know more then one language and design more then one project in various language. The diamond shape operator here shows association of programmer with both project and language.

- ✓ **Aggregation**

Objects which are composed of other objects are known as aggregations. They may involve containment (the contained objects are components of the larger object) or containership .

Overview

Containment v Containers

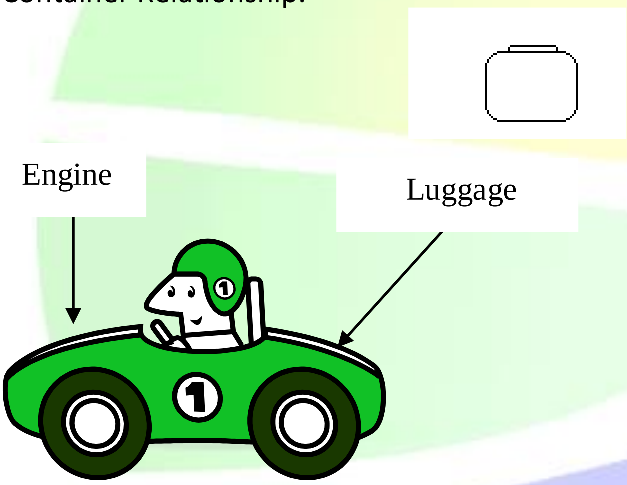
A commonly used term for this type of class relationship is containment' but there is a semantic difference between this term and the idea of container classes' which also may contain objects of other classes, but do not depend on them for their representation. There fore, although they can both be seen as types of aggregation, we should draw an important distinction between containment and containers, as follows:

1. In ‘containment` a composition hierarchy defines how an object is composed of other objects in a fixed relationship. The aggregate object cannot exist without its components, which will probably be of a fixed and stable number, or at least will vary within a fixed set of possibilities.
2. A container is an object (of a container class) which is able to contain other objects. The existence of the container is independent of whether it actually contains any objects at a particular time, and contained objects will probably be a dynamic and possibly heterogeneous collection (i.e. the objects contained may be many different classes).

Contrasts containment and a container using the example of a car:-

A Car Engine, gear, clutch etc. is the fixed component of car, without these components a car is not complete car so this type of component relationship is fixed in nature and called Containment Relationship.

On other Hand if a car contain luggage then this relationship is called variable relationship because without luggage a car is a complete car thus this represent
Container Relationship.



What is the key difference between `containment and a container?

Containment implies that the aggregate object is made up of components –without the components the object could not exit. Containers exist independently of their contents-they simply able to contain other objects.

Properties of aggregations

Rumbaugh [Rumbaugh et al, 1991p.37] notes that there are certain properties asso- ciated with the in an aggregation,

1. Transitivity : if A is a part of B and B is part of C, then A is part of C eg: If patio doors are part of lounge, and lounge is part of house, then patio doors are part of house.
2. Antisymmetry: if A is part of B, then B is not part of A eg If kitchen is part of house, then house is not part of kitchen
3. Propagation : The environment of the part is the same as that of the assembly eg If thecar is in the garage then the steering wheel is unlikely be somewhere else (if it is then car seems likely to stay in the garage!)

Layers of aggregation

Aggregation may well exist several layers so that objects are composed off component objects which themselves are composed of other objects. A train, for example, is composed of one or more locomotives and a number of coaches or wagons. The locomotives and coaches/wagons are composed of many smaller components.

The objects which comprise parts of a larger object may not have an exis-tence independent of the larger object.

Aggregation can be fixed, variable or recursive:

1. Fixed the particular number and types of the component parts are predefined eg a car has one engine, four wheels, one steering wheel etc.
2. Variable : the number of levels of aggregation are fixed, but the number of parts may vary (like the train)
3. Recursive: the object contains components of its own type, a Russian doll (each one containing a smaller doll) A more specific example in C++ is the ability for an object to contain a pointer of its own type allowing it to send messages to other objects of the same class.

✓ **Benefits of Oops**

- Through inheritance, we can eliminate redundant code and extend the use of existing
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of object to co-exits without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implement able form.
- Object-oriented systems can be easily upgraded from small to large systems .
- Message passing techniques for communication between objects makes the interface description with external systems much simpler.
- Software complexity can easily manage.

✓ **Disadvantage of Oops**

(i) **Resource demands –** since an object oriented program can require a much greater processing overhead than one written using traditional methods, it work more slowly this not a good from a customer’s point of view.

(ii) **Object persistence –**an object natural environment is in RAM as a dynamic entity. This is in contrast to traditional data storage in files or databases where the natural environment of the data is on external storage. This causes problem when we want objects to persist between runs of a program, even more so between different applications.

(iii) **Complexity** –the message passing between many objects in a complex application can be difficult to trace and debug.

(iv) **Reusability –** when inheritance is used, it is not easy to produce reusable objects between applications, since it makes their class closely coupled to the rest of the hierarchy. Object can become application specific to reuse with inheritance. It is extremely difficult to link together different hierarchies, making it difficult to coordinate very large systems. In most object oriented language, the introduction of inheritance severely compromises encapsulation

✓ Object Oriented Languages

Object-oriented programming is not the right of any particular language. Like structured programming , OOP concepts can be implemented using languages such as C and Pascal .however, programming becomes clumsy and may generate confusion when the programs grow large, A language that is specially designed to support the OOP concepts makes it easier to implement them.

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories,

1. Object-based programming languages, and
2. Object-oriented programming languages.

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object-based programming are:

- data encapsulation
- data hiding and access mechanisms
- automatic initialization and clear-up of objects
- operator overloading

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

Object-oriented programming incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statement:

Object-based features +inheritance +dynamic binding.

Languages that Support these feature include C++, SmallTalk, ObjectBase
Pascal and Java.

There are a large number of object base and object-oriented programming language.

✓ Types of object

There are four types of object (or in fact any other data type) which we may instantiate in program.

External (global) objects

An external object is one which is persistent and visible throughout a program module i.e. its scope is an entire module (source file) it may also be made visible in other modules. Objects which fall into the category of external would be ones whose number and identities remained constant throughout an application.

Automatic objects

As well as external global objects we may also have a number of locally declared Automatic objects –objects which exist in a predictable manner for a particular period of time. The key difference between an external and automatic object is that whereas an automatic objects instantiated within the scope of part of a program module, an external object is in instantiated outside of any scope (in C++ scope is defined by braces) `automatic` objects are automatically destroyed when they fallout of the scope in which they were instantiated .

Static objects

External objects are persistent and visible throughout the lifetime of a program, whereas automatic objects are only persistent and visible within the scope in which they are declared. There is also the possibility in C++ to explicitly declare a variable or object which has the scope (in terms of visibility) of an automatic objects but the lifetime of an external object. This is known as a 'static object'

Difference

External objects exist for the lifetime of the program and their visibility is global. Automatic objects exist as long as they remain in scope and are visible only within that scope. Static objects are created and visible within a particular scope, but persist from their point of creation until the end of the program.

Dynamic Objects

When we cannot predict the identities, and lifetime of the objects which will be represented at run time, then they cannot be uniquely named is (i.e. they cannot be external, static or automatic.) if objects are unpredictable, then they must be represented dynamically.

1. Creating and destroying dynamic objects using the `new` and delete` operators,

```
Employee *E=new Employee;
```

2. Calling the methods of dynamic objects using the `arrow` operator (->)

```
 E->getValues();
```

3. Destroying Objects

```
   delete E;
```

The Life Time of Named Objects.

1. Automatic objects

Objects instantiated inside the local scope of a function or other structure with its body defined by braces. They only exist while they are in scope.

This object get destroyed when controls comes out function body or block or a sub block.

Example:

```
void main()
{
Employee E1, E2; //an automatic object
…
}
```

2. external objects

Objects instantiated outside any function body. These have `file scope` and exist for the lifetime of the program.

This object doesn't destroyed when controls comes out function body or block or a sub block.

Example:

```
Employee E1, E2; //an external object
void main()
{
…
}
```

3. Static objects

Objects instantiated inside local scope and having local visibility, but persisting from their declaration to the end of the program.

Example :

```
void main()
{
static Employee E1,E2; //a static  object
…}
```

✓ **Exception Handling**

We know that it is very rare that a program works correctly first time. It might have bugs. The two most common types of bugs are logic errors and syntax errors. The logic errors occur due to poor understanding of the problem and solution procedure. The syntactic errors arise due to poor understanding of the language itself. We can detect these errors by using exhaustive debugging and testing procedures.

We often come across some peculiar problems other than logic or syntax errors. They are known as exceptions. Exceptions are run time error or unusual conditions that a program may encounter while executing. Run time Error might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space. When a program encounters as exceptional condition, it is important that identified and dealt with effectively run time errors.

Exception handing was not part of the original C++ it is new feature added to ANSI C++.

Today, almost all compilers support this feature.

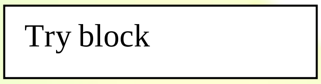
Exceptions are of two kinds, namely, synchronous exceptions and asynchronous. Errors such as “out-of-range index” and “over-flow” belong to the synchronous type exceptions. The errors that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions. The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions.

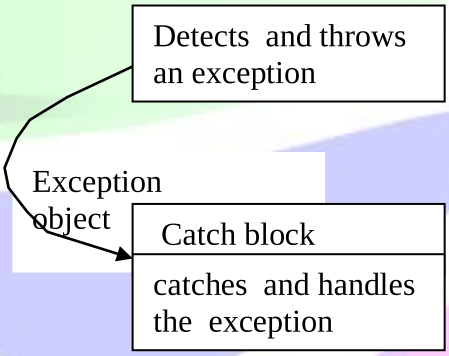
The mechanism suggests the following tasks:

1. Find the problem (hit the exception).
2. Inform that as errors has occurred (throw the exception).
3. Receive the errors information (catch the exception).
4. Take corrective actions (handle the exception).

The errors handling code basically consists of two segments, one to detect errors and to throw exceptions, and the other to catch the exceptions and to the take appropriate actions.

C++ exceptions handling mechanism is basically built upon three keywords, namely, try, throw, and catch the keyword try is used to preface a block of statements is known as try block. When as exception is detected, it is thrown using a throw statement in the try block. A catch block defined by the keyword catch ‘catches’ the exception ‘thrown’ by the try block, and handles it appropriately. The relationship is shown in fig:




Detects and throws an exception
Exception object
Catch block
catches and handles the exception

The catch block that catches as exception must immediately follow the try block that throws the exception, the general form of these two block are as follows.

```
…..
….. try
{
        …..
        throw   exception                    // block of statements which
        …..                                  // detect and throws an exception
        …..
}
catch (type arg)                        // catches exception
{
        .....
        .....                           // block of statements that
        .....                           // handles the exception
        .....
}
.....
....
```

When the try block throws an exception, the program control leaves the try block and enters the catch statement of the catch block where user can customize the error message.

- ✓ **Benefits of inheritance**

1. Subclasses provide specialized behaviors from the basic of common elements provided by the super class. Through the use of inheritance, programmers can reuse the code in the super class many times.
2. Programmers can implement super classes called abstract classes that define “generic” behaviors. The abstract super class define and may partially implement the behaviors, but much of the class is undefined and unimplemented. other programmers fill in the details with specialized subclasses.

Inheritance offers many useful features to programmers, Classes are easily maintainable and understandable when set relationship between class using inheritance.

- ✓ **Identifying Objects and Classes**

Objects can often be identified in term of the real world objects as well as the abstract objects. Therefore, the best place to look for objects is the application itself.

To construct the object model we must identify the relevant object and classes from the application domain. Object includes physical entities like houses, employee and machine. While identifying class we must emphasis on logical relationship, association and aggregations.

Analysis of relationships between the classes is central to the structure of a system. Therefore, it is important to identify appropriate classes to represent the object in the solution space and establish their relationships. The major relationships that are important in the context of design are:

1) Inheritance Relationships
2) Containment Relationship
3) Use Relationship(Aggregation)

Identifying Objects

There are three helpful ways you identify objects when you are designing the system.

1) A Checklist kind of objects
   a) External entities
   b) Things
   c) Occurrences of Events
   d) Roles
   e) Organizational units
   f) Places
   g) Structures

2) Grammatical parse of a piece of text describing the problem and outline Solution

   In the grammatical parse you select the nouns and noun Phrases as the potential objects and verbs as possible operation performed or by the objects.

   Like Bank Customer1, Customer2;

   Customer text represents Bank Customer.

3) Potential Objects
   a) Retained Information- the objects needs to remember information.
   b) Needed Service- the object has operations which changes its attributes.
   c) Common Attributes- all occurrences of an object have the attributes.
   d) Common Operation- all occurrences of an object have the operations.
   e) Essential Requirements-External entities, which produce, consume Information.
   f) Multiple Attributes- holds a state of object.

Identifying Right Classes

Redundant Classes: if two classes express the same information then it must be generalized.

Irrelevant Classes: if a class has little or nothing to do with problem. It should be eliminated.

Vague classes: a class should be specific.

Attribute: A class contains attribute which define the state of object. Ex: age, name, salary etc.

Operations: Most important part of class contains logical abstraction about object or we can say that operation that applied on object.

Roles: The name of the class that reflect the nature and role of the class that plays an association

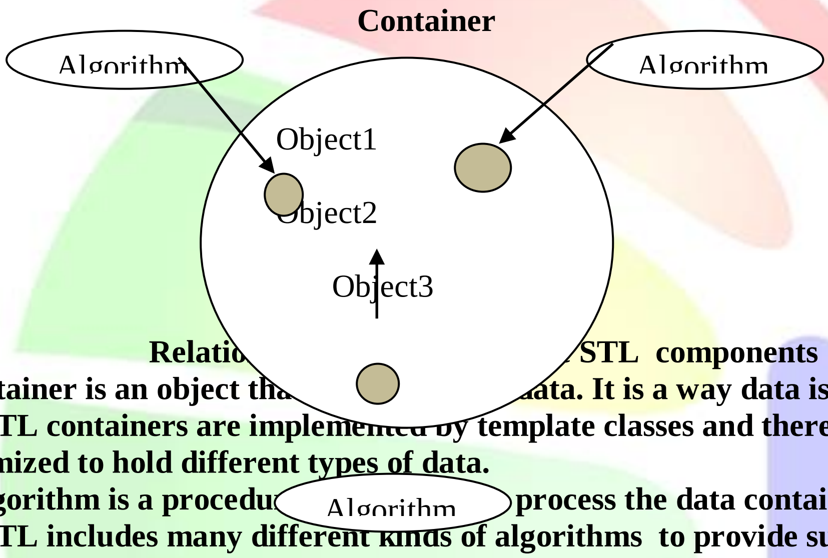
Introduction

Template could be used as a standard approach for storing and processing of data. The collection of these generic classes and function is called the Standard Template Library (STL). The STL has now become a part of ANSI standard C++ class library.

COMPONENTS OF STL

The STL contains several components. But at its core are three key components. They are:
Containers,
Algorithms, and
iterators.

These three components work in conjunction with one another to provide support to a variety of programming solution. The relationship between the three components is shown in Fig Algorithms employ iterators to perform operation stored in containers.



Relationship between the three STL components

A container is an object that actually stores data. It is a way data is organized in memory. The STL containers are implemented by template classes and therefore can be easily customized to hold different types of data.

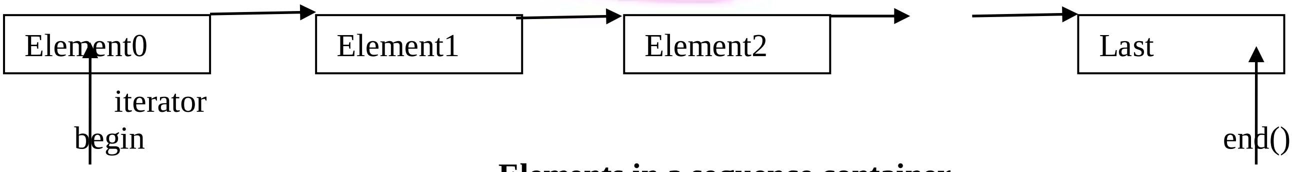
An algorithm is a procedure that is used to process the data contained in the containers. The STL includes many different kinds of algorithms to provide support to tasks such as initializing, searching, copying, sorting, and merging. Algorithms are implemented by template function.

An iterator is an object (like pointer) that points to an element in a container. We can use iterators to move through the contents of containers. Iterators are handled just like pointers.

We can increment or decrement them. Iterators connect algorithms with containers and play a key role in the manipulation of data stored in the containers.

Sequence Containers

Sequence containers store elements in a linear sequence, like a line as shown in Fig. Each element is related to other element by its position along the line. They all expand themselves to allow insertion of elements and all of them support a number of operation on them.



Elements in a sequence container

Associative Containers

Associative containers are designed to support direct access to elements using keys. they are not sequential. There are four types of associative container:

Set

Multiset
Map
Multimap

All these containers store data in a structure called tree which facilitates fast searching, deletion, and insertion. However, these are very slow for random access and inefficient for sorting.

Derived Containers

The STL provider three derived containers namely, **stack, queue,** and **priority _queue** These are also known as container adaptors. Stacks, queues and priority queue can be created from different sequence containers. The derived containers do not support iterators and therefore we cannot use them for data manipulation. However, they support two member functions **pop()** and **push()** for implementing deleting and inserting operations.

ALGORITHMS

Algorithms are functions that can be used generally across a variety of containers for processing their contents. Although each container provides functions for its basic operations, STL provides more than sixty standard algorithms to support more extended or complex operations. Standard algorithm also permit us to work with two different type of containers at the same time. Remember, STL algorithms are not member functions or friends of containers. They are standalone template function.

STL algorithms, based on the nature of operations they perform, may be categorized as under.

- Retrieve or non – mutating algorithms
- Mutating algorithms
- Sorting algorithms
- Set algorithms
- Relational algorithms

Table Non – mutating algorithms

Operations	Description
Count()	Counts occurrence of a value in a sequence
Equal()	True if two ranges are the same
Find()	Finds first occurrence of a value in a sequence
Search()	Finds a subsequence within a sequence

Table Mutating algorithms

Operations	Description
copy()	Copies a sequence
fill()	Fills a sequence with a specified value
remove()	Deletes elements of a specified value
replace()	Replaces elements with a specified value
reverse()	Reverses the order of elements

Table Sorting algorithms

Operations	Description

binary()	Conducts a binary search on an ordered sequence
lower _bound()	Finds the first occurrence of a specified value
merge()	Merges two sorted sequences
pop _heap()	Deletes the top element
push _heap()	Adds an element to heap
sort()	Sorts a sequence
upper _bound()	Finds the last occurrence of a specified value

Table Set algorithms

Operations	Description
includes()	Finds whether a sequence is a subsequence of another
set _difference()	Constructs a sequence that is the difference of two ordered sets
set _intersection()	Constructs a sequence that contains the intersection of ordered sets
set _symmetric difference()	Produces a set which is the symmetric difference between two ordered set
set _union()	Produces sorted union of two ordered sets

Table Relation algorithms

Operations	Description
equal()	Finds whether two sequences are the same
max()	Gives maximum of two value
min()	Gives minimum of two value

ITERATORS

Iterators behave like pointers and are used to access container elements. They are often used to traverse from one element to another, a process known as iterating through the container.

There are five types of iterators as described in Table

Table Iterators and their characteristics

Iterator	Access method	Direction of movement	I/0 capability	Remark
Input	Linear	Forward only	Read only	Cannot be saved
Output	Linear	Forward only	Write only	Cannot be saved
Forward	Linear	Forward only	Read/Write	Can be saved
Bidirectional	Linear	Forward and backward	Read/Write	Can be saved
Random	Random	Forward and backward	Read/Write	Can be saved

Functionality **Venn diagram of iterators**

The input and output iterators support the least functions. They can be used only to traverse in a container. The forward itrerator support all operations of input and output itrerators and also retains its position in the container. A bidirectional itreator, while supporting all forward itrerator operations, provides the ability to move in the backward direction in the container. A random access iterator combines the functionality of a bidirectional iterator with an ability to jump to an arbitrary location. Table sum marizes the operation that can be performed on each iterator type.

Table Operations supported by iterators

Iterator	Element access	Read	Write	Increment operation	comparison
Input	->	v =*p		++	==, !=
Output			*p = v	++	
Forward	->	v =*p	*p = v	++	==, !=
Bidirectional	->	v =*p	*p = v	++, --	==, !=
Random access	->,[]	v =*p	*p = v	++, --, +. -,+=, -=	==, !=, <,>,<=,>=

Vectors

The **vector** is the most widely used container. It stores elements in contiguous memory locations and enable direct access to any element using the subscript operator[]. A **vector** can change its size dynamically and therefore allocates memory as needed at run time.

The **vector** container supports random access iterators, and a wide range of iterator operations See may be applied to a **vector** iterator. Class **vector** supports a number of constructors for creating **vector** objects.

Table Important member functions of the vector class

Function	Task
at()	Give a reference to an element
back()	Give a reference to the last element
begin()	Give a reference to the first element
capacity()	Give the current capacity of the vector
erase()	Deletes specified element
insert()	Insert element in the vector
size()	Give the number of elements

Lists

The **list is** another container that is popularly used. It supports a bidirectional, linear list and providers an efficient implementation for deletion and insertion operations. Unlike a vector, which supports random access, a list can be accessed sequentially only.

Table Important member functions of the list class

Function	Task

begin()	Give reference to the first element
clear()	Deletes all the elements
end()	Give reference to the end of the list
insert()	Insert elements as specified
remove()	Removes elements as specified
reverse()	Reverses the list
sort()	Sort the list

The metaclass

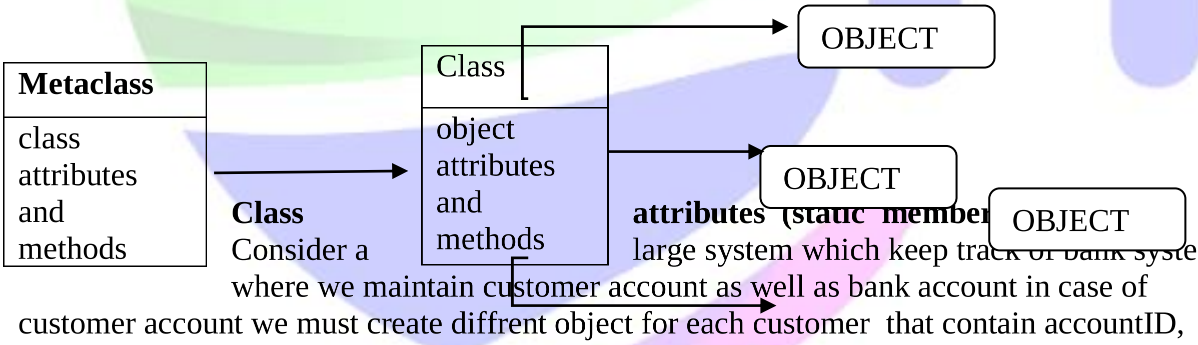
The use of ‘meta’ as a prefix has a wide range of meanings, including, simply, ‘about’, the term ‘metaclass’ is similer in usage to ‘metalanguage’- a language used to describe some other language (ie the metalanguage tells us about the language).

There are also a number of terms where ‘meta’ is used to imply some form of abstraction (such as ‘metaphysics’). From these interpretations we may conclude that the metaclass tells us about the class. And may be seen as an abstraction of the class in the same way that the class (as an ‘abstract data type’) is an abstraction of a set of object. The metaclass is often described as the ‘class of a class’, the class, as we know, holds the attributes and methods which will apply to objects of the class – it is the class of the objects.

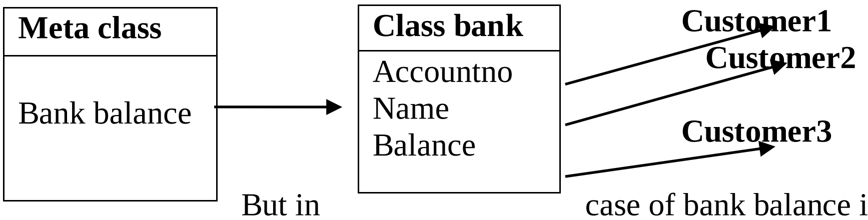
The metaclass hold the attributes and methods which will apply to the class itself – therefore it is the class of the class ! (see figure).

Every class has one (and only one) metaclass. And the meta class contain those parts of a class which are not appropriate to be duplicated for every specific object.

The metaclass may seem a rather esoteric concept(mysterious), but it is basically a repository for those part of a class which must exist at the run time of a program(objects).

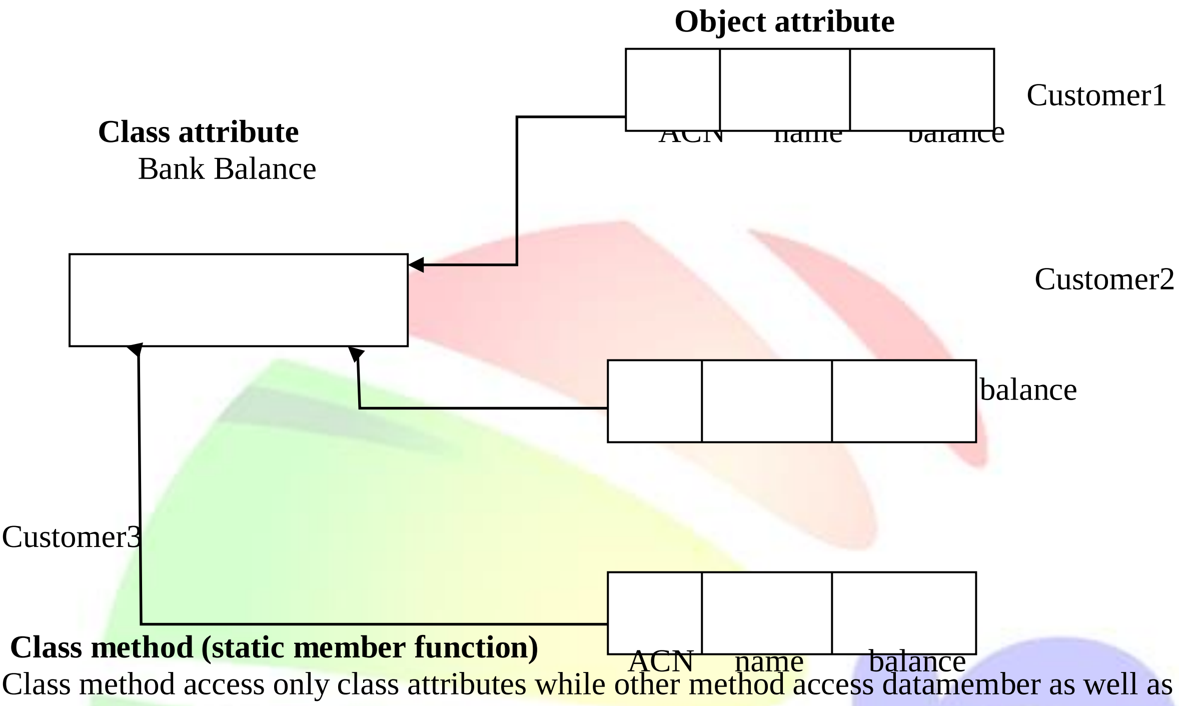


Class attributes (static member Consider a large system which keep track of bank system where we maintain customer account as well as bank account in case of customer account we must create diffrent object for each customer that contain accountID, customer name and its balance.



But in case of bank balance it is not part of customer, but each time when customer deposite or withdraw amount it must be updated. so there two type of attributes define within class, ***object*** ***attributes and class attributes*** in C++ ,object attributes are data member of class while class attributes are define within class preceding

with keyword static.
Class attributes create singal location in memory and it shared by all the object of the same class.For Example bankbalance. In following figure there are three customer object sahred variable bankbalance.



Class method (static member function)
Class method access only class attributes while other method access datamember as well as class attribute.
Ex. Static Data member/function(Bank).

Persistent objects, streams and files

Object which only exist while a program is running are known us transient object –means a transient object is one whose life time does not extend beyond a single program run.
Example of transient Data Automatic(local),Global, Static Objects.
Those whose life times extends beyond the boundaries of a single program run are known us persistence object means these type of object stored there data on disk.
There are two ways of approaching the implementation of persistent objects.

1) Stored Object Data in files using traditional file based approach, and again reload in object to process data.
2) Use an Object Oriented Database

Traditional file system is complex approach to maintain data, because it holds only object data in a file and it doesn't store structure/schema of class, thus it is very difficult for application to identify structure of data file.
But in case of object oriented database data is stored along with attributes name in structured way known as Schema, so it is very easy to handle data in Object Oreinted Database Management System(ODBMS) because it provide easiest way to insert, read, update and delete information from database using built in query Known as SQL.
(Oracle, Sybase, ingress is example of ODBMS).
Example of file handling for persistent object(insertion and display of employee data).

What is UML?

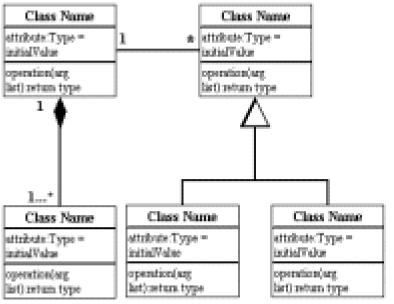
UML stands for Unified Modeling Language. This object-oriented system of notation has evolved from the work of Grady Booch, James Rumbaugh, Ivar Jacobson, and the Rational Software Corporation. These renowned computer scientists fused their respective technologies into a single, standardized model. Today, UML is accepted by the Object Management Group (OMG) as the standard for modeling object oriented programs.

Types of UML Diagrams

UML defines nine types of diagrams: class (package), object, use case, sequence, collaboration, statechart, activity, component, and deployment.

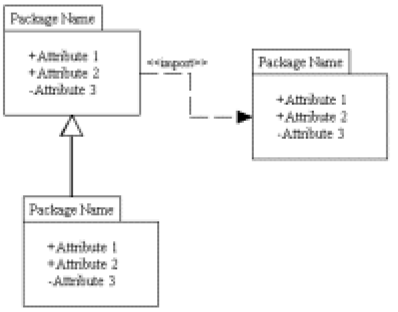
Class Diagrams

Class diagrams are the backbone of almost every object oriented method, including UML. They describe the static structure of a system.


Class Name
attribute:Type = initialValue
operation(arg list):return type
1
*
1
1...*

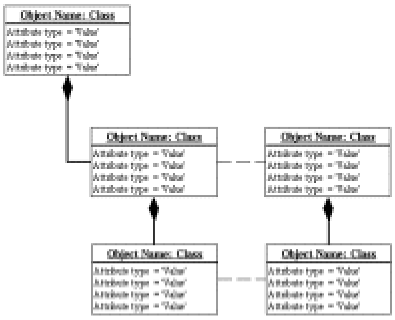
Package Diagrams

Package diagrams are a subset of class diagrams, but developers sometimes treat them as a separate technique. Package diagrams organize elements of a system into related groups to minimize dependencies between packages.



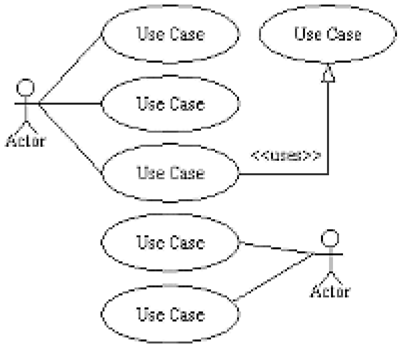
Object Diagrams

Object diagrams describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy.



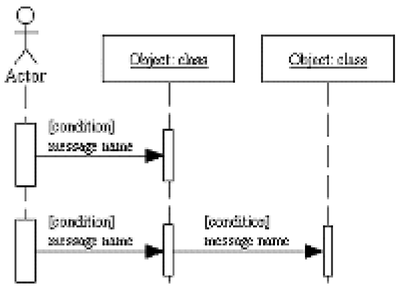
Use Case Diagrams

Use case diagrams model the functionality of system using actors and use cases.



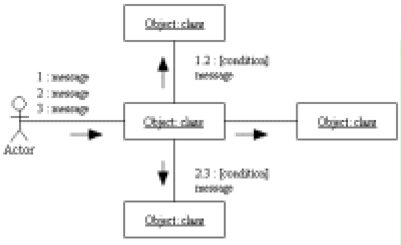
Sequence Diagrams

Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.



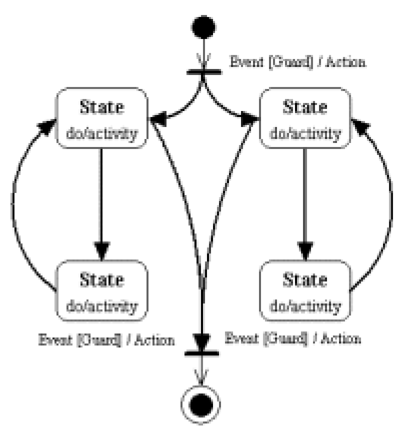
Collaboration Diagrams

Collaboration diagrams represent interactions between objects as a series of sequenced messages. Collaboration diagrams describe both the static structure and the dynamic behavior of a system.



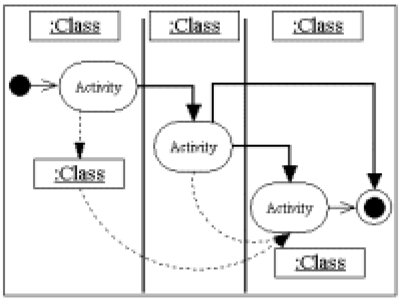
Statechart Diagrams

Statechart diagrams describe the dynamic behavior of a system in response to external stimuli. Statechart diagrams are especially useful in modeling reactive objects whose states are triggered by specific events.



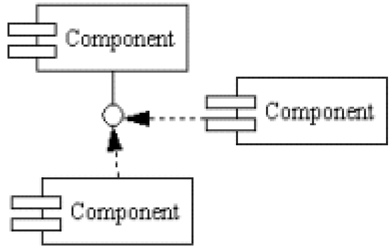
Activity Diagrams

Activity diagrams illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operation.



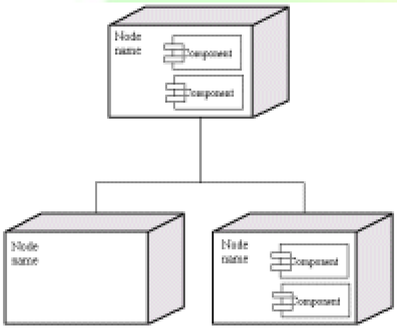
Component Diagrams

Component diagrams describe the organization of physical software components, including source code, run-time (binary) code, and executables.



Deployment Diagrams

Deployment diagrams depict the physical resources in a system, including nodes, components, and connections.



Object Oriented Analysis and Design

Object-oriented analysis (OOA) refers to the methods of specifying requirements of the software in terms of real-world objects, their behavior, and their interaction . Object-oriented (OOD), on the other hand turns the software requirements into specification for objects and derives class hierarchies from which the objects can be created. Finally, *Object oriented programming (OOP)* refers to the implementation of the program using objects, in an object-oriented programming language such
as C++.

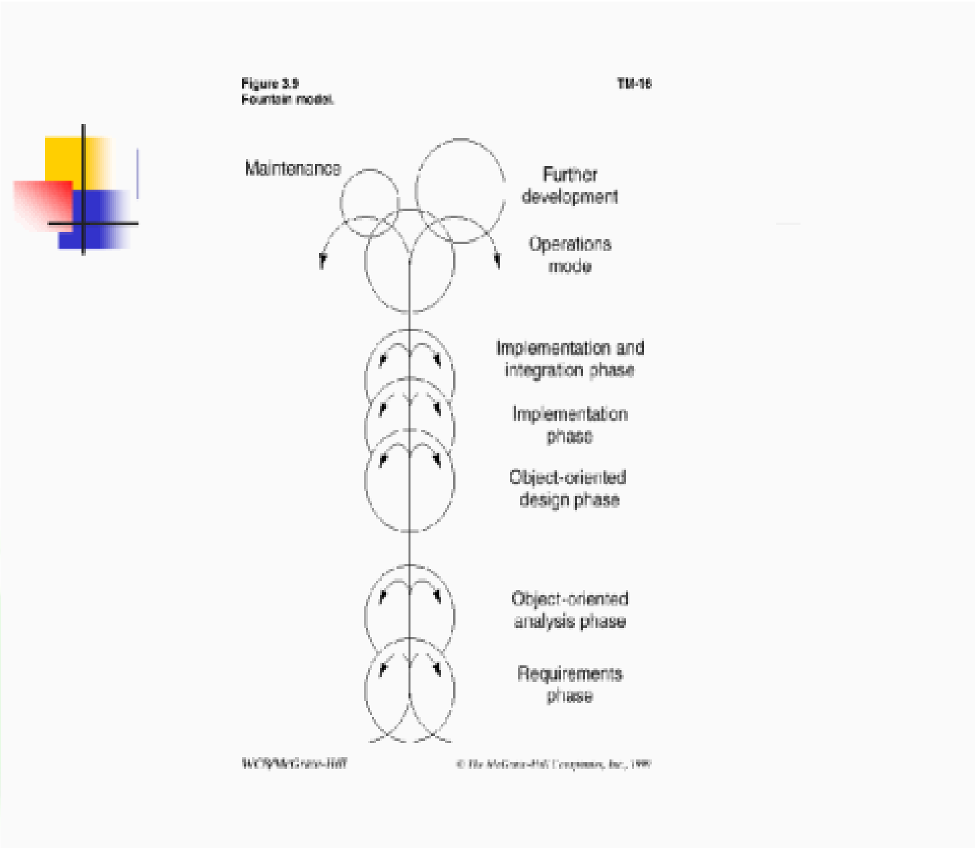


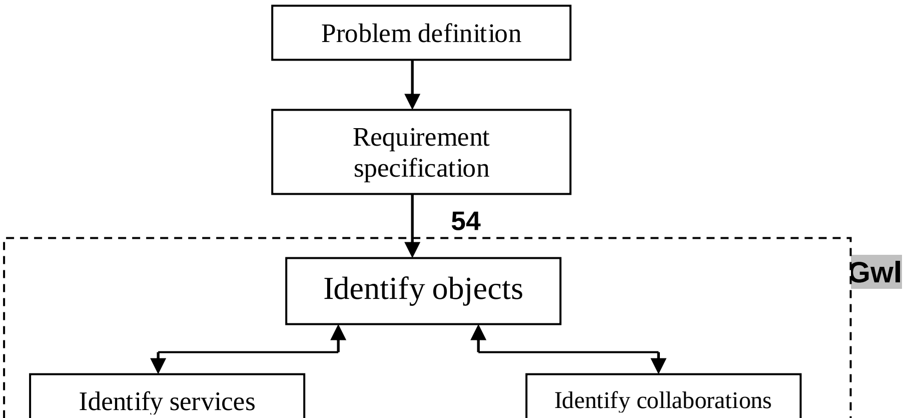
Figure 3.9
Fountain model.

TM-16
Maintenance
Further development
Operations mode
Implementation and integration phase
Implementation phase
Object-oriented design phase
Object-oriented analysis phase
Requirements phase
WCB/McGraw-Hill
© The McGraw-Hill Companies, Inc., 1999

OBJECT-ORIENTED ANALYSIS

Object-oriented analysis provides us with a simple ,yet powerful, mechanism for identifying objects, the building block of the software to be developed. The analysis is basically concerned with the decomposition of a problem into its component parts and establishing a logical model to describe the system function .
The object-oriented analysis (OOA) approach consists of the following steps:

1. Understanding the problem.
2. Drawing the specifications of requirements of the user and the software.
3. Identifying the objects and their attributes.
4. Identifying the services that each object is expected to provide (interface).
5. Establishing inter-connections (collaborations) between the objects in terms of services required and services rendered.



Problem Understanding

The first step in the analysis process is to understated the problem of the use. The problem statement should be refined and redefined in terms of computer system engineering that could suggest a computer-based solution. The problem statement should be stated, as far as possible, in a single, grammatically correct sentence. This will enable the software engineers to have a highly focused attention on the solution of the problem statement provides the basis for drawing the requirements specification of both the user and the software .

Requirements Specification

Once the problem is clearly defined, the next step is to understand what the proposed system is required to do. It is important at this stage to generate a list of user requirements. A clear understanding should exist between the user and the developer of what is required .based on the user requirements, the specifications for the software should be drawn. The developer should state clearly:

- What outputs are required.
- What processes are involved to produce these outputs.
- What resources are required .

These specifications often serve as a reference to test the final product for its performance of the intended tasks.

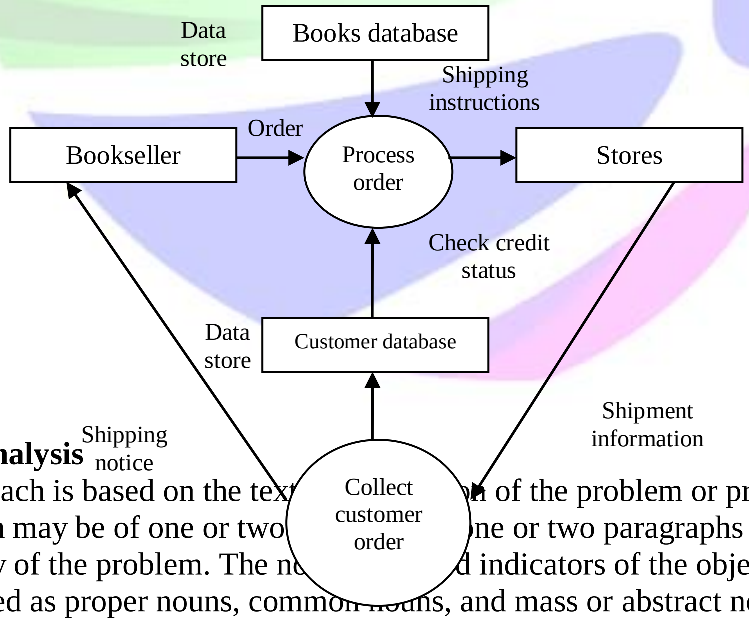
Identification of objects

Objects can often be identified in terms of the real-world objects as well as the abstract objects. Therefore, the best places to look for objects is the application itself. The application may be analyzed by using one of the following two approaches:

1. Data flow diagrams (DFD)
2. Textual analysis (TA)

Data flow diagram

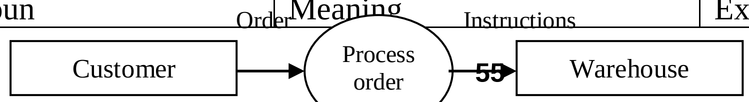
The application can be represented in the form of a data flow diagram indicating how the data moves from one point to another in the system. The boxes and data stores in the data flow diagram are good candidates for the objects. The process bubbles correspond to the procedures. Illustrates a typical data flow diagram. It is also known as a data flow graph or a bubble chart .



Textual analysis

This approach is based on the text n of the problem or proposed solution. The description may be of one or two ne or two paragraphs depending on the type and complexity of the problem. The n d indicators of the objects. The names can further be classified as proper nouns, common ns, and mass or abstract nouns. Table shows the various types of nouns and their meaning.

Type of noun	Meaning	Example



Common noun	Describe classes of things (entites)	Vehicle, customer income, deduction
Proper noun	Names of specific things	Maruti car, john, ABC company
Mass or abstract noun	Describe a quality, quantity or an activity associated with a noun	Salary-income house-loan, feet traffic

Identification of Services

Once the objects in the solution space have been identified, the next step is to identify a set of services that each object should offer. Services are identified by examining all the verbs and verb phrases in the problem description statement. Verbs which can note actions or occurrences may be classified as shown in Table.

Types of verb	Meaning	Examples
Doing verbs	Operations	Read, get, display, buy
Being verbs	Classifications	Is an, belongs to
Having verbs	Composition	Has an, is part of
Compare verbs	Operations	Is less than, is equal to
Stative verbs	Invariance-condition	To be present, are owned

Object Oriented Design

Design is concerned with the mapping of objects in the problem space into objects in the solution space, and creation an overall structure (architectural model) and computational models of the system. This stage normally uses the bottom-up approach to build the structure of the system and the top=down functional decomposition approach to design the class member functions that provide services. It is particularly important to design the class member functions that provide services. It is particularly important to construct structured hierarchies, to identify abstract classes, and to simplify the inter-object communications.

1. Review of objects created in the analysis phase.
2. Specification of class dependencies.
3. Organization of class hierarchies.
4. Design of classes.
5. Design of member functions.
6. Design of driver program.

Review of problem space objects

An exercise to review the objects identified in the problem space is undertaken as a first step in the design stage. The main objective of this review exercise is to refine the objects in terms of their attributes and operations and to identify other objects that are solution specific. Some guidelines that might help the review process are.

1. If only one object is necessary for a service (or operation), then it operates only on that object .
2. If two or more objects are required for an operation to occur, then it is necessary to identify which object’s private part should be known to the operation .
3. If an operation requires knowledge of more than one type of objects, then the operation is not functionally cohesive and should be rejected.

Applying these guidelines will help us refine the services of the objects. Further, the redundant or extraneous objects are removed, synonymous services are combined and the names of the operations(function) are improved clearly the kind of processing involved.

Class Dependencies

Analysis of relationships between the classes is central to the structure of a system. Therefore, it is important to identify appropriate classes to represent the objects in the solution space and establish their relationships. The major relationships that are important in the context of design are:

1. Inheritance relationships.
2. Containment relationships.
3. Use relationships.

Inheritance relationship is the highest relationship that can be represented in C++. It is a powerful way of representing a hierarchical relationship directly. The real appeal and power of the inheritance mechanism is that it allows us to reuse a class that is almost, but not exactly, what we want and to tailor the class in a way that it does not introduce any unwanted side effects into the rest of the class. We must review the attributes and operations of the classes and prepare an inheritance relationship table as shown in Table

Class	Depends on
A	….
B	A
C	A
D	B
B1	B
B2	B

Containment relationship means the use of an object of a class as a member of another class. This is an alternative and complimentary technique to use the class inheritance. But, it is often a tricky issue to choose between the two techniques. Normally, if there is a need to override attributes or functions, then the inheritance is the best choice, On the other hand, if we want to represent a property by a variety of types, then the containment relationship is the right method to follow. Another place where we need to use an object as a member is when we need to pass an attribute of a class as an argument to the constructor of another class. The "another" class must have a member object that represents the argument. The inheritance represents is_a relationship and the containment represents has_a relationship.

Use relationship gives information such as the various classes a class uses and the way it uses them. For example, a class A can use classes B and C in several ways:

- A reads a member of B.
- A call a member of C.
- A creates B using new operator.

Organization of Class Hierarchies

In the previous step, we examined the inheritance relationships. We must re-examine them and create a class hierarchy so that we can reuse as much data and/or functions that have been designed already. Organization of the class hierarchies involves identification of common attributes and functions among a group of related classes and then combining them to form a new class. The new class will serve as the super class and the others as subordinate classes (which derive attributes from the super class). The new class may or may not have the meaning of an object by itself. If the object is created purely to combine the common attributes, it is called an abstract class.

Design of Classes

We have identified classes, their attributes, and minimal set of operations required by the concept a class is representing. Now we must look at the complete details that each class represents. The important issue is to decide what functions are to be provided. For a class to be useful, it must contain the following functions, in addition to the service functions;

1. Class management functions.

    - How an object is created?
    - How an object is destroyed?

2. Class implementation functions.

   What operations are performed on the data type of the class?
3. Class access functions.

   How do we get information about the internal variables of the class?
4. Class utility functions.

   How do we handle errors?

Design of Member Functions

We have so far identified

1. Classes and identified,
2. Data members,
3. Interfaces,
4. Dependencies, and
5. Class hierarchy (structure).

It is time now to consider the design of the member functions. The member functions define the operations that are performed on the object's data. These functions behave like any other C function and therefore we can use the top-down functional decomposition technique to design them.

C++ Templates

C++ templates are a powerful mechanism for code reuse, as they enable the programmer to write code that behaves the same for data of any type.

Templates in C++ programming allows function or class to work on more than one data type at once without writing different codes for different data types. Templates are often used in larger programs for the purpose of code reusability and flexibility of program.

Templates are also known as generic functions or classes which are used to implement a generic structure for many structures of the same statements with different data-types.

Templates are expanded by the compiler in as many definitions as the function calls of the different types used.

We can declare a template with the keyword template. Like so:

```
template<typename T>
template<class T>
```

The template keyword tells the compiler that what follows is a template, and that T is a template parameter that identifies a type.

There are two types of templates:

- Function templates
- Class templates

Function templates

```
#include <iostream.h>
template <class T>
T square(T x)
{   T result;
   result = x * x;
   return result;
}
void main()
{   int    i=2, ii;
   float  x=2.2, xx;
   double y=2.2, yy;
```

```
  ii = square<int>(i);
  cout << i << ": " << ii << endl;

  xx = square<float>(x);
  cout << x << ": " << xx << endl;

  // Explicit use of template
  yy = square<double>(y);
  cout << y << ": " << yy << endl;

  // Implicit use of template
  yy = square(y);
  cout << y << ": " << yy << endl;}
```

Class templates

```
#include<iostream.h>
#include<conio.h>
template <class T>
class TwoNum
{ private:
  T x,y;
  public:
  void getValues()
  {
  cout<<"Enter Two Numbers:";
  cin>>x>>y;
  }
  void putValues()
  {
  cout<<x<<","<<y<<endl;
  }

 T add()
  {  T z;
    z=x+y;  } };
void main()
{Twonum <int>A;
 A.getValues();
 A.putValues();
 int j=A.add();
 cout<<j<<endl;

Twonum <double>B;
 B.getValues();
 B.putValues();
 double k=B.add();
 cout<<k<<endl;  }
```

Friend Classes

It is often useful for one class to see the private variables of another class, even though these variables should probably not be made part of the public interface that the class supports.

A friend class in C++ can access the "private" and "protected" members of the class in which it is declared as a friend.

```
#include<iostream.h>
 #include<conio.h>
 class readint
 {
```

```
float a,b;
public:
void read()
{
cout<<"\n\nEnter the First Number : ";
cin>>a;
cout<<"\n\nEnter the Second Number : ";
cin>>b;
}
friend class sum;
};
class sum
{
public:
float c;
void add(readint rd)
{
c=rd.a+rd.b;
cout<<"\n\nSum="<<c;
}
};
void main()
{
int cont;
readint rd;
sum s;


clrscr();
rd.read();
s.add(rd);

getch();
}
```

Console I/O Operations

Every program takes an input from the user, process it then outputs the user. The entire process of input and output happens on the Console of the system. This process is known as Console Input/Output.

Unformatted Console I/O Operations:

Unformatted I/O opreration generally used to read data from keyboard,memory or file or use to write data in memory,variable or file.
The general functions which are used for displaying I/O as discusses earlier are ‘<<’ and ‘>>’ with cout and cin. Some of the other functions used are listed below:

1. get() and put()

- get() is a member function of istream class.
- put is a member function of ostream class.
- get(char *) -> reads single character from the keyboard and assigns to an argument.
- Allow whitespaces & new line character

Example:

```
char   c;
    cin.get( c); // reads and assigns to c
  cout.put(c); //display character
```

2. getline() and write()

getline()

- getline() function reads the line of text and reading ends when it reaches new line character.
- Allow whitespaces

- Syntax :
- cin.getline(line ,size);
- where **'line'** is array of character and size is maximum number of characters.
- **Ex: char x[30]**
  cin.getline(x,30); //read string thru keyboard

  write()
- The write() method displays entire line on the screen.
- The write methods reads the character by character untill it reaches the new line.
- Syntax :
- cout.write(line ,size);
- Where line is character array and size is number of character to print.
- **Ex: char x[]="Gwalior";**
  Cout.write(x,2);// output will be Gw

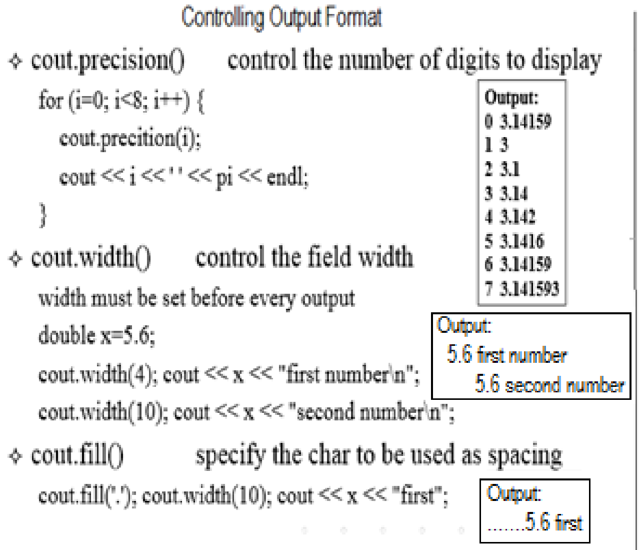
Formatted Console I/O Operations:

C++ have number of formatting outputs, this includes:

1. ios class
2. manipulators
3. user-defined

IOS Functions

Access function	**Effect**	**Default**
`width()`	Minimal field width	0
`precision()`	Precision of floating point values	6
`fill()`	Fill character for padding	The space character
`setf(flag,bitfield)`	Set formatting flags	
`unsetf(bitfiled)`	Remove the set flag	



Controlling Output Format

- cout.precision() control the number of digits to display

```
for (i=0; i<8; i++) {
   cout.precition(i);
   cout << i << ' ' << pi << endl;
}
```

```
Output:
0 3.14159
1 3
2 3.1
3 3.14
4 3.142
5 3.1416
6 3.14159
7 3.141593
```

- cout.width() control the field width

width must be set before every output

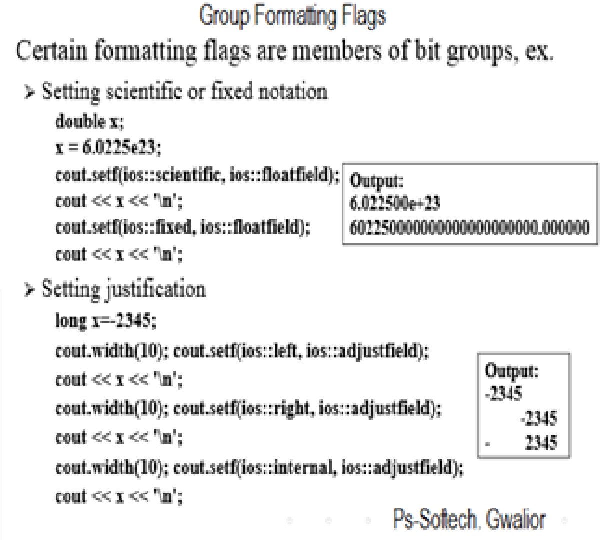
```
double x=5.6;
cout.width(4); cout << x << "first number\n";
cout.width(10); cout << x << "second number\n";
```

```
Output:
 5.6 first number
       5.6 second number
```

- cout.fill() specify the char to be used as spacing

```
cout.fill('.'); cout.width(10); cout << x << "first";
```

```
Output:
.......5.6 first
```



Group Formating Flags

Certain formatting flags are members of bit groups, ex.

- Setting scientific or fixed notation

```
double x;
x = 6.0225e23;
cout.setf(ios::scientific, ios::floatfield);
cout << x << '\n';
cout.setf(ios::fixed, ios::floatfield);
cout << x << '\n';
```

```
Output:
6.022500e+23
602250000000000000000000.000000
```

- Setting justification

```
long x=-2345;
cout.width(10); cout.setf(ios::left, ios::adjustfield);
cout << x << '\n';
cout.width(10); cout.setf(ios::right, ios::adjustfield);
cout << x << '\n';
cout.width(10); cout.setf(ios::internal, ios::adjustfield);
cout << x << '\n';
```

```
Output:
-2345
     -2345
-     2345
```

Ps-Softech. Gwalior

Flags and their effects on operators

Format flag	**Group**	**Effect**	**Default**
left right internal	***adjustfield***	left right adds fill characters at designated internal point	
dec	***basefield***	decimal base	
oct	***basefield***	octal base	
hex	***basefield***	hexadecimal base	
fixed	***floatfield***	in fixed-point notation	
scientific	***floatfield***	in scientific notation	

Manipulators(iomanip.h)

Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer’s choice of display.There are numerous manipulators available in C++. Some of the more commonly used manipulators are provided here below:

Manipulator	Effect	Equivalent
`Endl`	Inserts newline and flushes buffer	`\n`
`resetiosflags (ios_base::fmtflags mask)`	Clears ***ios*** flags	`unsetf()`
`setfill(charT c)`	Sets fill character for padding	`fill()`
`setiosflags (ios_base::fmtflags mask)`	Sets ***ios*** flags	`setf(mask)`
`setprecision (int n)`	Sets precision of floating point values	`precision(n)`
`setw(int n)`	**Sets** minimal field width	`width(n)`

Example:
Cout<<setw(10)<<setfill(‘#’)<<567;
Output
#######567

Cout<<setw(10)<<setfill(‘#’)<<setiosflags(ios::left)<<)567;
Output
567#######

Creating Our Own Manipulator