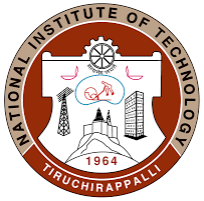


CSPC31: Principles of Programming Languages

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
Ph: 999 470 4853
E-Mail: balakrishnan@nitt.edu

Books

- **Text Books**
  - ✓ **Robert W. Sebesta, *"Concepts of Programming Languages"*, Tenth Edition, Addison Wesley, 2012.**
  - ✓ Michael L. Scott, *"Programming Language Pragmatics"*, Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
  - ✓ Allen B Tucker, and Robert E Noonan, *"Programming Languages – Principles and Paradigms"*, Second Edition, Tata McGraw Hill, 2007.
  - ✓ R. Kent Dybvig, *"The Scheme Programming Language"*, Fourth Edition, MIT Press, 2009.
  - ✓ Jeffrey D. Ullman, *"Elements of ML Programming"*, Second Edition, Prentice Hall, 1998.
  - ✓ Richard A. O'Keefe, *"The Craft of Prolog"*, MIT Press, 2009.
  - ✓ W. F. Clocksin, C. S. Mellish, *"Programming in Prolog: Using the ISO Standard"*, Fifth Edition, Springer, 2003.

Chapters

Chapter No.	Title
1.	Preliminaries
~~2.~~	~~Evolution of the Major Programming Languages~~
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
~~5.~~	~~Names, Binding, Type Checking and Scopes~~
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
~~11.~~	~~Abstract Data Types and Encapsulation Constructs~~
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages

Chapter 12 – Support for Object-Oriented Programming

Objectives

- Introduction to object-oriented programming
- Discussion of the primary design issues for inheritance and dynamic binding
- Support for object-oriented programming in C++
- Overview of the implementation of dynamic bindings of method calls to methods in object-oriented languages

Micsellaneous

Object-Oriented Programming Concepts

- Encapsulation
- Abstract Data Types
- Inheritance
- Polymorphism (Dynamic Binding)

Miscellaneous

```
function add(int c, int d)
{
    return (c + d);
}

void main()
{
   int a = 5, b = 10;
   printf("%d", add(a, b));
}
```

O/P: 15

```
class Add
{
  public:
     int num1, num2;
     int sum(int n1, int n2)
     {
       return n1+n2;
     }
}
int main()
{
   //Creating object of class
   Add obj;
   obj.num1 = 5;
   obj.num2 = 10;
   cout << sum(num1, num2);
}
```

O/P: 15

```
class Add
{
  protected:
     int num1, num2;
     int sum(int n1, int n2)
     {
       return n1 + n2;
     }
}
int main()
{
   //Creating object of class
   Add obj;
   obj.num1 = 5;
   obj.num2 = 10;
   cout << sum(num1, num2);
}
```

O/P: Error

Miscellaneous

```
class construct
{
public:
   int a, b;

   construct()
   {
      a = 10;
      b = 20;
   }
};

int main()
{
   // Default constructor called automatically  when the object is created
   construct c;
   cout << "a: " << c.a << endl
        << "b: " << c.b;
   return 1;
}
```

O/P: a: 10 b: 20

```
class Point
{
private:
   int x, y;

public:
   Point(int x1, int y1)
   {
      x = x1;
      y = y1;
   }

   int getX()
   {
      return x;
   }
   int getY()
   {
      return y;
   }
};
```

```
int main()
{
   // Constructor called
   Point p1(10, 15);

   // Access values assigned by Constructor
   cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

   return 0;
}
```

O/P: p1.x = 10, p1.y = 15

Types

- Default Constructor
- Parameterized Constructor

Miscellaneous

```
class HelloWorld
{
  public:
     HelloWorld()
     {
        cout<<"Constructor is called"<<endl;
     }
     ~HelloWorld()
     {
       cout<<"Destructor is called"<<endl;
     }
     void display()
    {
      cout<<"Hello World!"<<endl;
    }
};
```

```
int main()
{
   HelloWorld obj;
   obj.display();
   return 0;
}
```

Destructor

A destructor is automatically called when:
1) The program finished execution
2) When a scope (the { } parenthesis) containing local variable ends
3) When you call the delete operator

O/P: Constructor is called
Hello World!
Destructor is called

Miscellaneous

Access Specifiers

- Public -> Members are accessible from outside the class
- Private -> Members cannot be accessed (or viewed) from outside the class
- Protected -> Members cannot be accessed from outside the class, however, they can be accessed in inherited classes

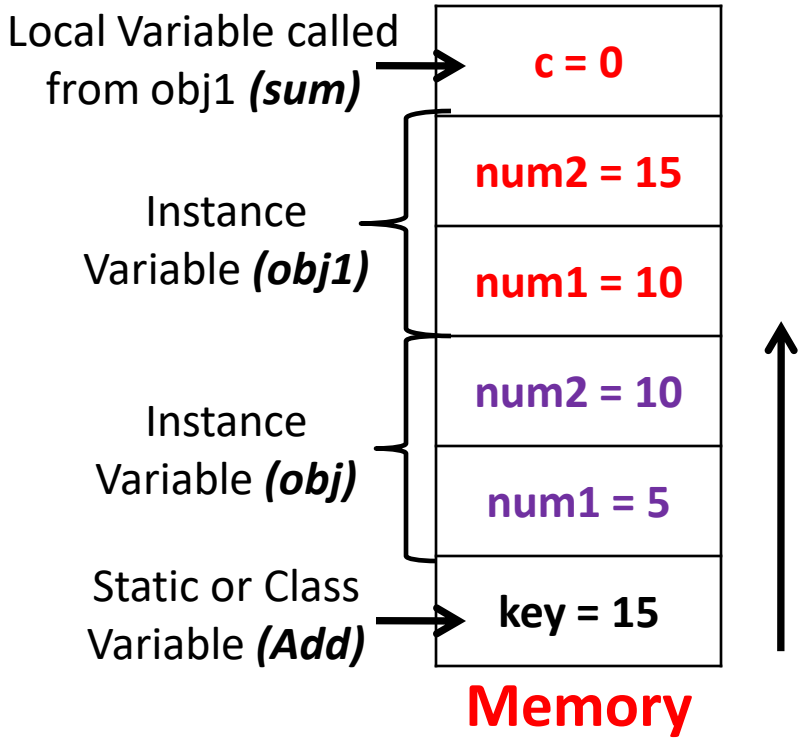
Miscellaneous

```
class Add
{
  public:
     int num1, num2;
     static int key;  // Default value = 0
     int sum(int n1, int n2)
     {
       int c = 0;
       return n1 + n2;
     }
}
int main()
{
   //Creating object of class
   Add obj, obj1;
   obj.num1 = 5;    obj1.num1 = 10;
   obj.num2 = 10;  obj1.num2 = 15;
   cout << obj1.sum(num1, num2);
}
int Add::key = 15;
```

Instance Variables

Static or Class Variable

Local Variable



Miscellaneous

```
//Base class
class Parent
{
   public:
     int id_p;
};

class Child : public Parent
{
   public:
     int id_c;
};
```

```
int main()
{
      Child obj1;
      obj1.id_c = 7;
      obj1.id_p = 91;
      cout << "Child id is " <<  obj1.id_c <<
endl;
      cout << "Parent id is " <<  obj1.id_p <<
endl;

      return 0;
  }
```

Inheritance -> Is Unidirectional

Types

- **Single Inheritance**
- Multiple Inheritance

O/P: Child id is 7
Parent id is 91

Miscellaneous

```
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

class FourWheeler
{
  public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle"
<< endl;
    }
};
```

```
class Car: public Vehicle, public
FourWheeler {

};

int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Inheritance

Types

- Single Inheritance
- **Multiple Inheritance**

O/P: This is a Vehicle
This is a 4 wheeler Vehicle

Miscellaneous

```
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

class FourWheeler
{
  public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle"
<< endl;
    }
};
```

```
class Car: public Vehicle, public
FourWheeler {
 public:
    Car()
    {
      cout << "This is a Car" << endl;
    }
};

int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Inheritance

Types

- Single Inheritance
- **Multiple Inheritance**

O/P: This is a Vehicle
This is a 4 wheeler Vehicle
This is a Car

Miscellaneous

Inheritance

```
class A
{
  public:
     int x;
  protected:
     int y;
  private:
     int z;
};

class B : public A
{
   // x is public
   // y is protected
   // z is not accessible from B
};
```

```
class C : protected A
{
   // x is protected
   // y is protected
   // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
   // x is private
   // y is private
   // z is not accessible from D
};
```

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Miscellaneous

```
class Geeks
{
   public:
   void func(int x)
   {
      cout << "value of x is " << x <<
endl;
   }

   void func(double x)
   {
      cout << "value of x is " << x <<
endl;
   }

   void func(int x, int y)
   {
      cout << "value of x and y is " << x
<< ", " << y << endl;
   }
};
```

```
int main()
{
   Geeks obj1;
   obj1.func(7);
   obj1.func(9.132);
   obj1.func(85, 64);
   return 0;
}
```

Polymorphism

When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments

Types

- **Functional Overloading (Compile Time)**
- Operator Overloading (Compile Time)
- Virtual Functions (Run Time)

Miscellaneous

- **Function Overloading (achieved at compile time)**
  - Provides multiple definitions of the function by changing signature
  - That is, changing number of parameters, change datatype of parameters (Return type doesn't play any role)
  - It can be done in base as well as derived class

    Example:

    void **area**(int a);

    void **area**(int a, int b);
- **Function Overriding (achieved at run time)**
  - Redefinition of base class function in its derived class with same signature
  - That is, return type and parameters
  - It can only be done in derived class

Function Overloading vs Function Overriding

Example:

```
Class a
{
 public: virtual void display()
 {
   cout << "hello";
 }
};
Class b:public a
{
  public: void display()
 {
   cout << "bye";
  }
};
```

https://www.geeksforgeeks.org/function-overloading-vs-function-overriding-in-cpp/

Miscellaneous

```
class Complex
{
   private:
      int real, imag;
   public:
    Complex(int r = 0, int i =0)
    {
       real = r;
       imag = i;
    }
    Complex operator + (Complex const
&obj)
    {
       Complex res;
       res.real = real + obj.real;
       res.imag = imag + obj.imag;
       return res;
   }
   void print()
   {
      cout << real << " + i" << imag << endl;
   }
};
```

```
int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2; // An example
call to "operator+"
   c3.print();
}
```

"+" -> Addition (Two Integers), Concatenation (Strings), Addition (Complex Numbers)

Types

- Functional Overloading (Compile Time)
- **Operator Overloading (Compile Time)**
- Virtual Functions (Run Time)

Miscellaneous

```
class base
{
public:
   virtual void print ()
   {
     cout<< "print base class" <<endl;
   }

   void show ()
   {
     cout<< "show base class" <<endl;
   }
};

class derived: public base
{
public:
   void print () //print () is already virtual function in derived class, we could also declared as virtual void print () explicitly
   {
     cout<< "print derived class" <<endl;
   }

   void show ()
   {
     cout<< "show derived class" <<endl;
   }
};
```

O/P: **print derived class**
show base class

```
int main()
{
   base *bptr;
   derived d;
   bptr = &d;

   //virtual function, binded at runtime (Runtime polymorphism)
   bptr->print();

   // Non-virtual function, binded at compile time
   bptr->show();

   return 0;
}
```

https://hownot2code.com/2017/08/10/c-pointers-why-we-need-them-when-we-use-them-how-they-differ-from-accessing-to-object-itself/

This type of polymorphism is achieved by Function Overriding. Function overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden

Types

- Functional Overloading (Compile Time)
- Operator Overloading (Compile Time)
- **Virtual Functions (Run Time)**

Miscellaneous

```
class base
{
public:
   void print ()
   {
     cout<< "print base class" <<endl;
   }

   void show ()
   {
     cout<< "show base class" <<endl;
   }
};

class derived: public base
{
public:
   void print ()
   {
     cout<< "print derived class" <<endl;
   }

   void show ()
   {
     cout<< "show derived class" <<endl
   }
};
```

```
int main()
{
   base *bptr;
   derived d;
   bptr = &d;

   //Non-virtual function, binded at compile time
   bptr->print();

   // Non-virtual function, binded at compile time
   bptr->show();

   return 0;
}
```

Note: "Virtual" keyword is not used. Hence, objects are binded at compile time itself

O/P: print base class
show base class

Miscellaneous

```
class base
{
public:
   void print ()
   {
     cout<< "print base class" <<endl;
   }

   void show ()
   {
     cout<< "show base class" <<endl;
   }
};

class derived: public base
{
public:
   void print ()
   {
     cout<< "print derived class" <<endl;
   }

   void show ()
   {
     cout<< "show derived class" <<endl
   }
};
```

```
int main()
{
    base bptr;
    bptr.print();
    bptr.show();
    derived d;
    d.print();
    d.show();

   return 0;
}
```

Note: Individual objects are created

O/P: **print base class**
show base class
print derived class
show derived class

Miscellaneous

```
class base
{
public:
   virtual void print ()
   {
     cout<< "print base class" <<endl;
   }

   void show ()
   {
     cout<< "show base class" <<endl;
   }
};

class derived: public base
{
public:
   void print () //print () is already virtual
function in derived class, we could also
declared as virtual void print () explicitly
   {
     cout<< "print derived class" <<endl;
   }

   void show ()
   {
     cout<< "show derived class" <<endl;
   }
};
```

```
int main()
{
    base bptr;
    bptr.print();
    bptr.show();
    derived d;
    d.print();
    d.show();

   return 0;
}
```

Note: In the case of individual objects, there is no effect even though virtual keyword is added

O/P: **print base class**
show base class
print derived class
show derived class

Miscellaneous

Abstract Method or Pure Virtual Method in C++

```
class Base
{
   int x;
public:
   virtual void fun() = 0;    // Pure Virtual Function
   int getX() { return x; }
};

// This class inherits from Base and implements
fun()
class Derived: public Base
{
   int y;
public:
   void fun()
   {
      cout << "fun() called";
   }
};
int main(void)
{
   Derived d;
   d.fun();
   return 0;
}
```

O/P: fun() called

Abstract Class (Interface) or Abstract Base Class in C++

```
class Test
{
   int x;
public:
   virtual void show() = 0;
   int getX() { return x; }
};

int main(void)
{
   Test t;
   return 0;
}
```

Such a class usually cannot be instantiated, because some of its methods are declared but are not defined (they do not have bodies). Any subclass of an abstract class that is to be instantiated must provide implementations (definitions) of all of the inherited abstract methods.

O/P: Compiler Error: cannot declare variable 't' to be of abstract type 'Test' because the following virtual functions are pure within 'Test': note: virtual void Test::show()

Miscellaneous

```
class Base
{
  int x;
public:
   virtual void fun() = 0;   //Pure Virtual Function
   virtual void fun1() = 0; // Pure Virtual Function
   int getX()
   {
     return x;
   }
};

class Derived: public Base
{
   int y;
public:
   virtual void fun() = 0;
   virtual void fun1() = 0;
   virtual void fun2() = 0;
};
```

```
Class Derived_1: public Derived
{
   int z:
   public:
    virtual void fun()
    {
      cout << "Show";
    }
     virtual void fun1()
    {
      cout << "Show_1";
    }
     virtual void fun2()
    {
      cout << "Show_2";
    }
};

int main(void)
{
   Base b;     // Throws Error
   Derived d; // Throws Error
   Derived_1 d;
   d.fun();
   return 0;
}
```

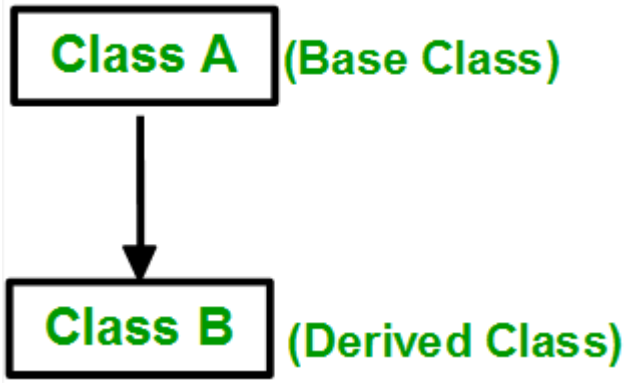
O/P: Show

Miscellaneous

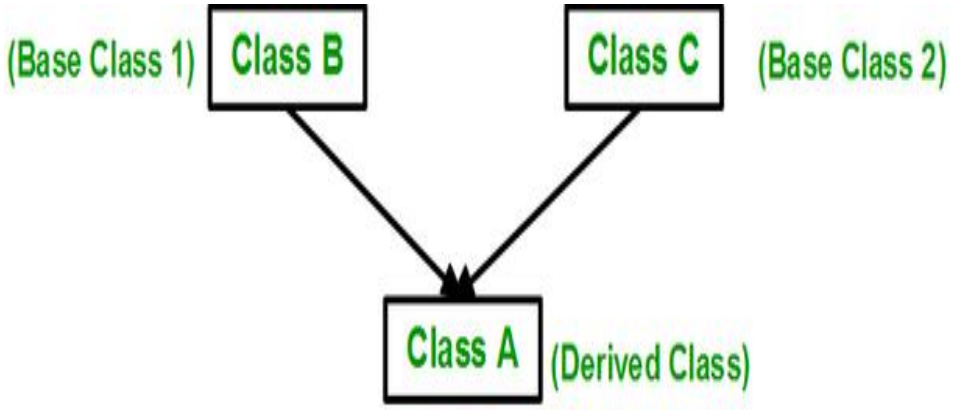
```
class MyClass {       // The class
      public:         // Access specifier
        void myMethod();   // Method/function declaration
    };

    // Method/function definition outside the class
    void MyClass::myMethod() {
      cout << "Hello World!";
    }

    int main() {
      MyClass myObj;     // Create an object of MyClass
      myObj.myMethod();  // Call the method
      return 0;
    }
```

Miscellaneous

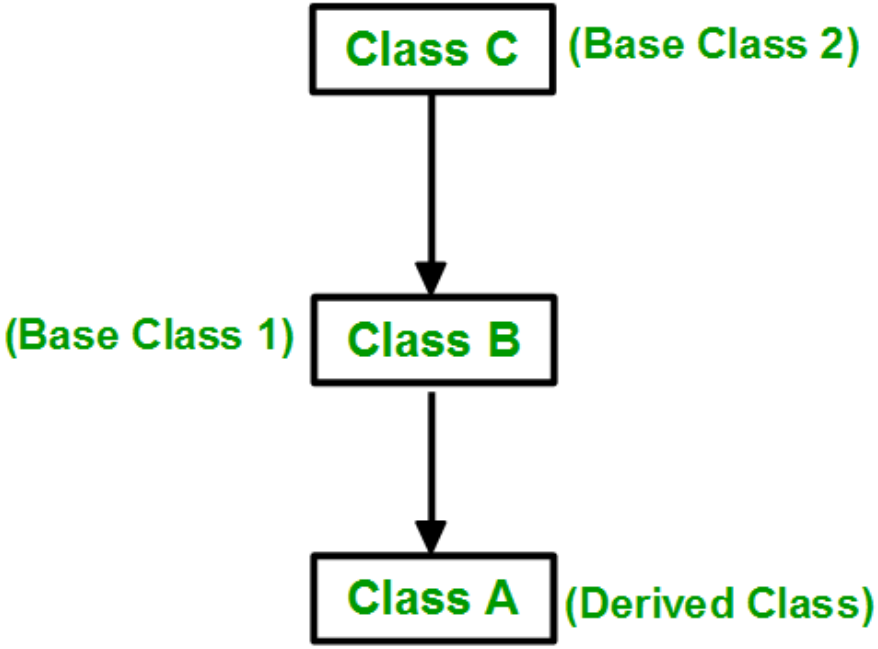
Single Inheritance



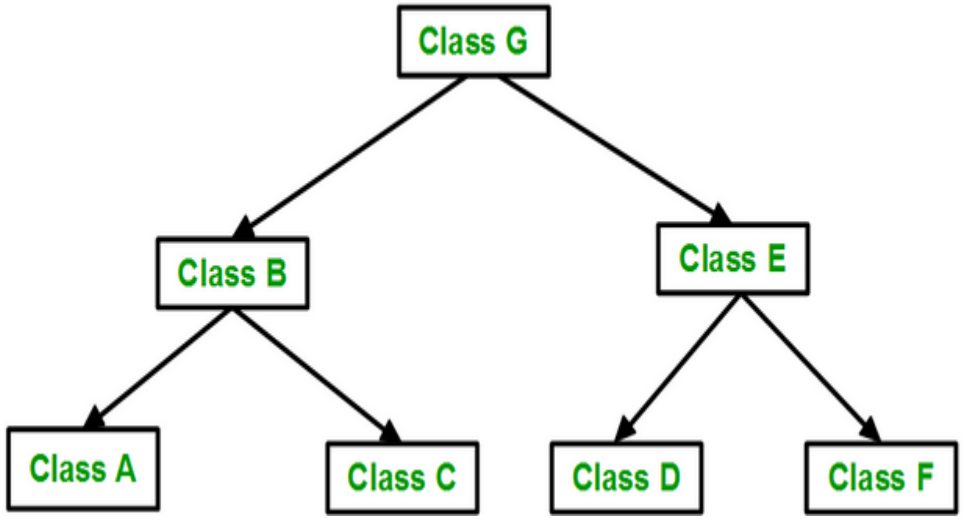
Multiple Inheritance



Multi-Level Inheritance

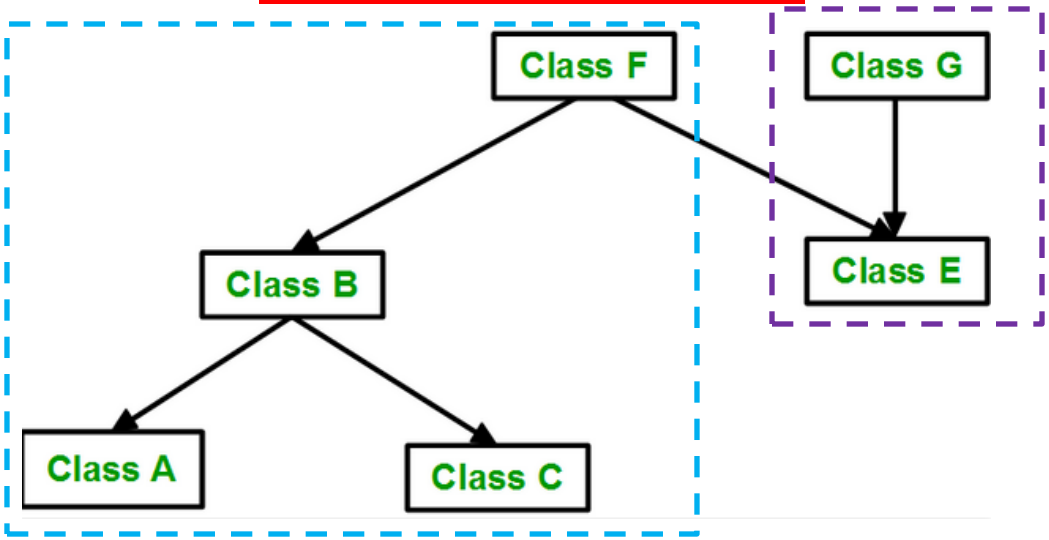


Hierarchical Inheritance

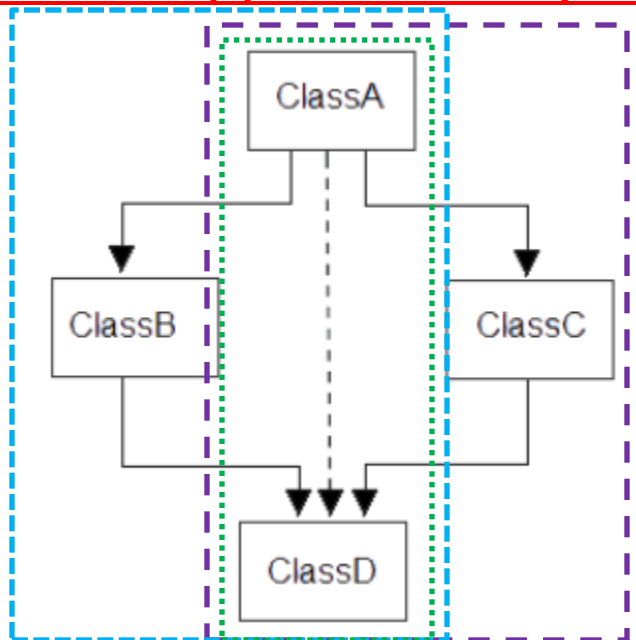


Miscellaneous

Hybrid (Virtual) Inheritance



Multipath Inheritance (Special Case of Hybrid Inheritance)



Miscellaneous

Friend Class

```
class Node
{
  private:
    int key;
    Node* next;
    /* Other members of Node Class */

    friend class LinkedList;
    // Now class  LinkedList can
   // access private members of Node
};

Class LinkedList
{
   public:
    int search()
    {
       .....
    }
};
```

Friend Function

```
class Node
{
   private:
      int key;
      Node* next;
  /* Other members of Node Class */

   friend int LinkedList::search();

   // Only search() of linkedList
   // can access internal members
};
```

- A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class
- Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:
  a) A method of another class
  b) A global function

Miscellaneous

```
class base
{
   public:
     base()
     {
        cout << "Base Class Constructor Called"
<< endl;
     }
     void print ()
     {
         cout<< "print base class" <<endl;
     }
     void show ()
    {
         cout<< "show base class" <<endl;
    }
    ~base()
    {
        cout << "Base Class Destructor Called"
<< endl;
    }
};
```

```
class derived: public base
{
   public:
      derived()
      {
         cout << "Dervied Class Destructor
Called" << endl;
      }
      void print ()
      {
         cout<< "print derived class" <<endl;
      }
      void show ()
      {
         cout<< "show derived class" <<endl;
      }
      ~derived()
      {
         cout << "Derived Class Destructor
Called" << endl;
      }
};
```

Destructor

https://hownot2code.com/2017/08/10/c-pointers-why-we-need-them-when-we-use-them-how-they-differ-from-accessing-to-object-itself/

Miscellaneous

```
int main()
{
    base bptr;
    bptr.print();
    bptr.show();
    derived d;
    d.print();
    d.show();
    cout << "1" <<endl;
    return 0;
    cout << "2" << endl;
}
```

O/P:
Base Class Constructor Called
print base class
show base class
Base Class Constructor Called
Derived Class Constructor Called
print derived class
show derived class
1
Derived Class Destructor Called — Object "d" deleted
Base Class Destructor Called — Object "d" deleted
Base Class Destructor Called — Object "bptr" deleted

```
int main()
{
    base *bptr;
    derived d;
    bptr = &d;
    bptr->print();
    bptr->show();
    cout << "1" << endl;
    return 0;
}
```

Destructor

O/P:
Base Class Constructor Called
Derived Class Constructor Called
print base class
show base class
1
Derived Class Destructor Called
Base Class Destructor Called

Miscellaneous

```
int main()
{
  base *bptr1 = new base();
  base *bptr;
  derived d;
  bptr = &d;
  bptr->print();
  bptr->show();
  cout << "1" << endl;
  return 0;
}
```

O/P:
Base Class Constructor Called
Base Class Constructor Called
Derived Class Constructor Called
print base class
show base class
1
Object "d" deleted → Derived Class Destructor Called
Base Class Destructor Called

Note: Object "bptr1" is not deleted

Destructor

```
int main()
{
  base *bptr1 = new base();
  delete bptr1;
  base *bptr;
  derived d;
  bptr = &d;
  bptr->print();
  bptr->show();
  cout << "1" << endl;
  return 0;
}
```

Note: Whenever you create an object in heap memory using "new" keyword, you have to delete it explicitly using "delete" keyword. Here, object "bptr1" is deleted explicitly

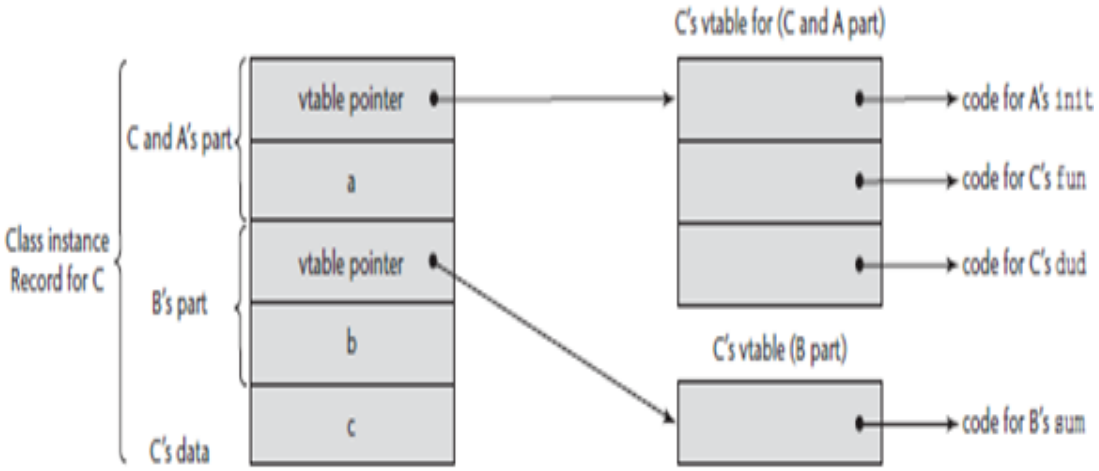
O/P:
Base Class Constructor Called
Base Class Destructor Called → Object "bptr1" deleted
Base Class Constructor Called
Derived Class Constructor Called
print base class
show base class
1
Object "d" deleted → Derived Class Destructor Called
Base Class Destructor Called

Miscellaneous

```
int main()
{
    base *bptr1;
    bptr1->print();
    bptr1->show();
    cout << "1" << endl;
    return 0;
}
```

O/P: print base class
show base class
1

Note: Neither constructor nor destructor are called in this case



Introduction

- All data are objects
- A language that is object oriented must provide support for three key language features -> Abstract Data Types, Inheritance and Dynamic Binding of method calls to methods

Inheritance

- Software reuse
- Abstract data types, with their encapsulation and access controls, are obviously candidates for reuse
    - The problem with the reuse of abstract data types is that, in nearly all cases, the features and capabilities of the existing type are not quite right for the new use
    - A second problem with programming with abstract data types is that the type definitions are all independent and are at the same level

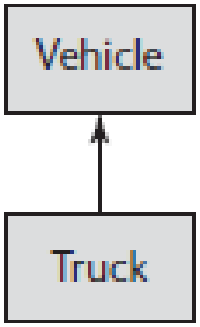
Introduction

- Inheritance offers a solution to both the modification problem posed by abstract data type reuse and the program organization problem
- Programmers can begin with an existing abstract data type and design a modified descendant of it to fit a new problem requirement
- Abstract data types in object-oriented languages -> Classes
- Instances of abstract data types (class instances) -> Objects
- A class that is defined through inheritance from another class -> Derived Class or Subclass
- A class from which the new class is derived is its parent class or superclass
- Subprograms that define the operations on objects of a class are called methods
- The calls to methods are sometimes called messages
- The entire collection of methods of an object is called the message protocol or message interface of the object

Introduction

- Computations in an object-oriented program are specified by messages sent from objects to other objects, or in some cases, to classes
- Passing a message is indeed different from calling a subprogram
  - A subprogram typically processes data that is either passed by its caller as a parameter or is accessed nonlocally or globally
    - A subprogram defines a process that it can perform on any data sent to it (or made available nonlocally or globally)
  - A message sent to an object is a request to execute one of its methods
    - At least part of the data on which the method is to operate is the object itself
    - Because the objects are of abstract data types, these should be the only ways to manipulate the object

Introduction



Variables -> Year, Color, Make

Variables -> Year, Color, Make, Capacity, Number of Wheels

- Several ways a derived class can differ from its parent
  - Parent class can define some of its variables or methods to have private access, which means they will not be visible in the subclass
  - Subclass can add variables and/or methods to those inherited from the parent class
  - Subclass can modify the behavior of one or more of its inherited methods
    - Modified method has the same name, and often the same protocol, as the one of which it is a modification
- New method is said to override the inherited method, which is then called an overridden method
- Purpose of an overriding method is to provide an operation in the subclass that is similar to one in the parent class, but is customized for objects of the subclass. **Eg:** Draw() method in Class Rectangle (Length, Width), Circle (Radius)

Introduction

- Classes can have two kinds of methods and two kinds of variables
- Commonly used methods and variables are called instance methods and instance variables
- Every object of a class has its own set of instance variables, which store the object's state
- Only difference between two objects of the same class is the state of their instance variables
- **Eg:** A class for cars might have instance variables for color, make, model and year
- Instance methods operate only on the objects of the class
- Class variables belong to the class, rather than its object, so there is only one copy for the class
- **Eg:** Counter to count the number of instances of a class -> The counter would need to be a class variable
- Class methods can perform operations on the class and possibly also on the objects of the class

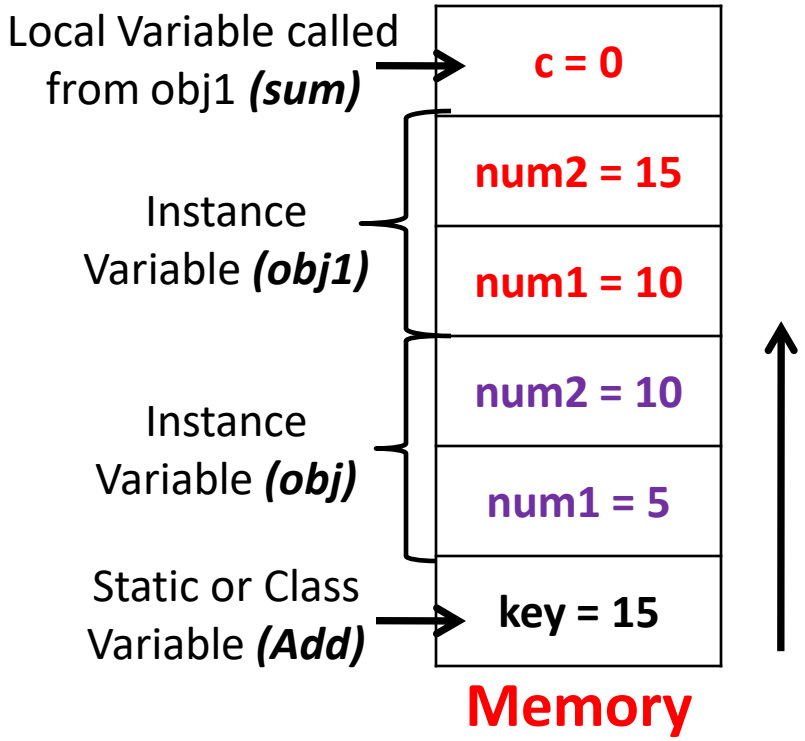
Miscellaneous

```
class Add
{
  public:
     int num1, num2;
     static int key;  // Default value = 0
     int sum(int n1, int n2)
     {
       int c = 0;
       return n1 + n2;
     }
}
int main()
{
   //Creating object of class
   Add obj, obj1;
   obj.num1 = 5;    obj1.num1 = 10;
   obj.num2 = 10;  obj1.num2 = 15;
   cout << obj1.sum(num1, num2);
}
int Add::key = 15;
```

Instance Variables

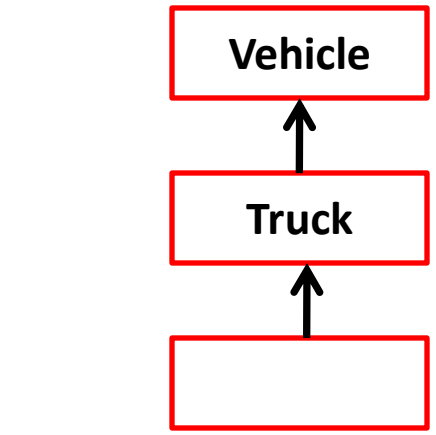
Static or Class Variable

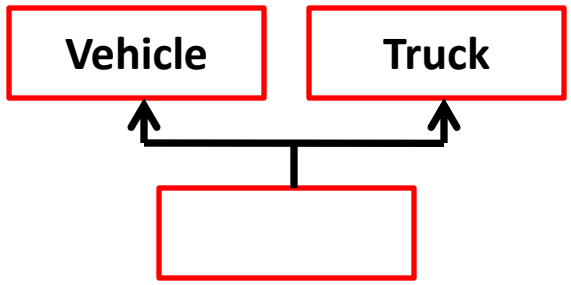
Local Variable



Introduction

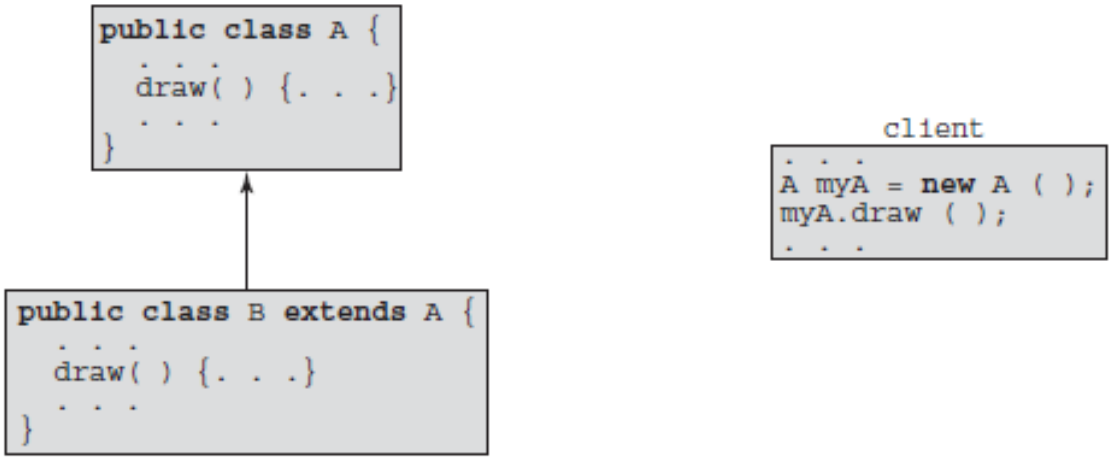
- If a new class is a subclass of a single parent class, then the derivation process is called single inheritance
- If a class has more than one parent class, the process is called multiple inheritance
- When a number of classes are related through single inheritance, their relationships to each other can be shown in a derivation tree
- The class relationships in a multiple inheritance can be shown in a derivation graph
- One disadvantage of inheritance is that it creates dependencies among the classes in an inheritance hierarchy
- This result works against one of the advantages of abstract data types, which is that they are independent of each other
- Of course, not all abstract data types must be completely independent
- But in general, the independence of abstract data types is one of their strongest positive characteristics
- However, it may be difficult, if not impossible, to increase the reusability of abstract data types without creating dependencies among some of them
- Furthermore, in many cases, the dependencies naturally mirror dependencies in the underlying problem space





Dynamic Binding

- Third characteristic (after abstract data types and inheritance) of object-oriented programming languages is a kind of polymorphism provided by the dynamic binding of messages to method definitions -> Dynamic Dispatch



- If a client of A and B has a variable that is a reference to class A's objects, that reference also could point at class B's objects, making it a polymorphic reference
- If the method draw, which is defined in both classes, is called through the polymorphic reference, the run-time system must determine, during execution, which method should be called, A's or B's (by determining which type object is currently referenced by the reference)

Miscellaneous

```
#include <iostream>
class Point
{
    int x, y, z;
  public:
    Point(int x, int y, int z)
    {
        this->x = x;
        this->y = y;
        this->z = z;
    }
    void display()
    {
        cout << "(" << x << ", " << y << ", "
     << z << ")" << endl;
    }
};
```

```
int main()
{
    Point p1(10, 15, 20);
    p1.display();
    Point *ptr;
    ptr = new Point(50, 60, 70);
    ptr->display();
}
```

O/P: (10, 15, 20)
(50, 60, 70)

- In C++, we can instantiate the class object with or without using the new keyword. If the new keyword is not used, then it is like normal object. This will be stored at the stack section. This will be destroyed when the scope ends. But for the case when we want to allocate the space for the item dynamically, then we can create pointer of that class, and instantiate using new operator.
- In C++, the new is used to dynamically allocate memory.

Miscellaneous

```
class Vehicle
{
   public:
     Vehicle()
     {
        cout << "This is a Vehicle" << endl;
     }
     virtual void draw()
     {
         cout<<"Vehicle Draw";
     }
};
```

O/P: This is a Vehicle
This is a 4 wheeler Vehicle
Fourwheeler Draw

```
class FourWheeler: public Vehicle
{
    public:
       FourWheeler()
       {
           cout << "This is a 4 wheeler
     Vehicle" << endl;
       }
       void draw()
       {
          cout<<"Fourwheeler Draw";
       }
};
void main()
{
    Vehicle *obj = new FourWheeler();
    obj->draw();
}
```

FourWheeler *obj = new Vehicle();
O/P: error: invalid conversion from 'Vehicle*' to 'FourWheeler*' [-fpermissive]

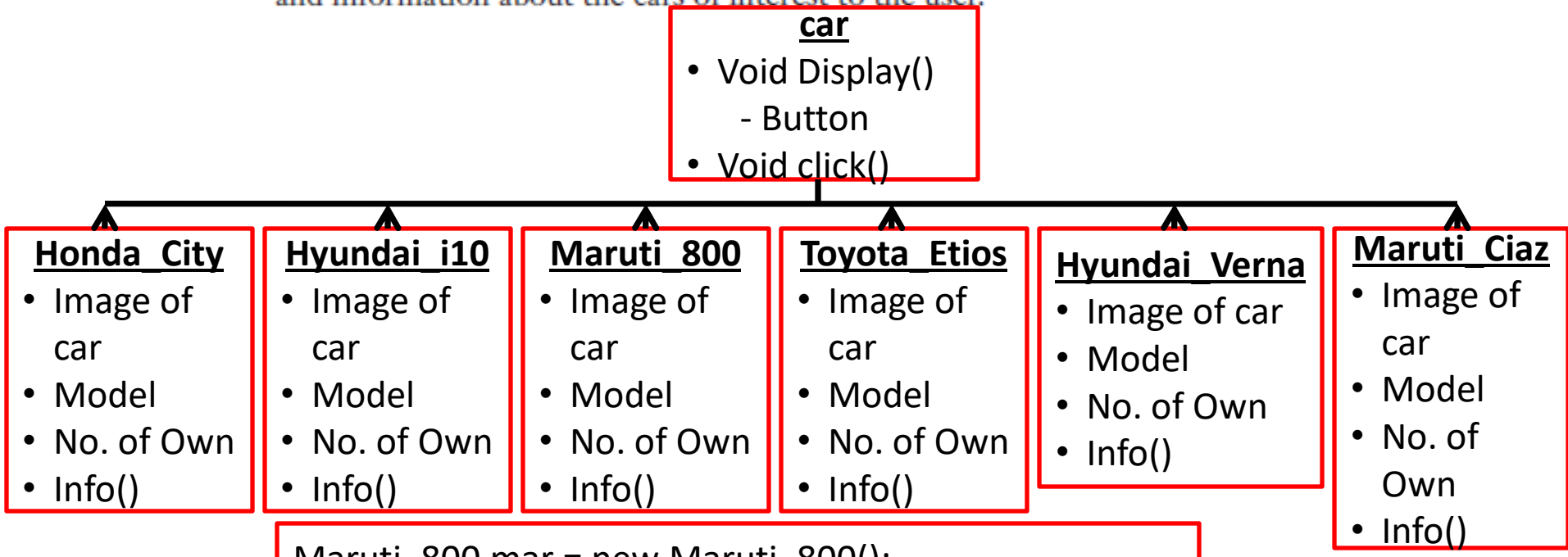
Dynamic Binding

- Polymorphism makes a statically typed language a little bit dynamically typed
  - Little bit is in some bindings of method calls to methods
- The type of a polymorphic variable is indeed dynamic
- One purpose of dynamic binding is to allow software systems to be more easily extended during both development and maintenance

Suppose we have a catalog of used cars that is implemented as a `car` class and a subclass for each car in the catalog. The subclasses contain an image of the car and specific information about the car. Users can browse the cars with a program that displays the images and information about each car as the user browses to it. The display of each car (and its information) includes a button that the user can click if he or she is interested in that particular car. After going through the whole catalog, or as much of the catalog as the user wants to see, the system will print the images and information about the cars of interest to the user. One way to implement this system is to place a reference to the object of each car of interest in an array of references to the base class, `car`. When the user is ready, information about all of the cars of interest could be printed for the user to study and compare the cars in the list. The list of cars will of course change frequently. This will necessitate corresponding changes in the subclasses of `car`. However, changes to the collection of subclasses will not require any other changes to the system.

Dynamic Binding

Suppose we have a catalog of used cars that is implemented as a `car` class and a subclass for each car in the catalog. The subclasses contain an image of the car and specific information about the car. Users can browse the cars with a program that displays the images and information about each car as the user browses to it. The display of each car (and its information) includes a button that the user can click if he or she is interested in that particular car. After going through the whole catalog, or as much of the catalog as the user wants to see, the system will print the images and information about the cars of interest to the user.




```
Maruti_800 mar = new Maruti_800();
Honda_City hon = new Honda_City();
Hyundai_i10 hyn = new Hyundai_i10(); car *obj[];
obj[0] =  mar; obj[1] = hon; obj[2] = hyn;
```

Design Issues for Object-Oriented Languages

- A number of issues must be considered when designing the programming language features to support inheritance and dynamic binding

Exclusivity of Objects

- A language designer who is totally committed to the object model of computation designs an object system that subsumes all other concepts of type
- Everything, from a simple scalar integer to a complete software system, is an object in this mind-set
- Advantage of this choice is the elegance and pure uniformity of the language and its use
- Disadvantage is that simple operations must be done through the message-passing process, which often makes them slower than similar operations in an imperative model, where single machine instructions implement such simple operations **// ADD B, A**
- In this purest model of object-oriented computation, all types are classes
- There is no distinction between predefined and user-defined classes
- All classes are treated the same way and all computation is accomplished through message passing

Design Issues for Object-Oriented Languages

- One alternative to the exclusive use of objects that is common in imperative languages to which support for object-oriented programming has been added is to retain the complete collection of types from a traditional imperative programming language and simply add the object typing model
  - Results in a larger language whose type structure can be confusing to all but expert users
- Another alternative to the exclusive use of objects is to have an imperative style type structure for the primitive scalar types, but implement all structured types as objects
  - Provides the speed of operations on primitive values that is comparable to those expected in the imperative model
  - Also leads to complications in the language
  - Nonobject values must be mixed with objects
  - Creates a need for so-called *wrapper classes* for the nonobject types, so that some commonly needed operations can be implemented as methods of the wrapper class

Design Issues for Object-Oriented Languages

Are Subclasses Subtypes?

- Issue here is relatively simple: Does an "is-a" relationship hold between a derived class and its parent class?
- If a derived class is a parent class, then objects of the derived class must expose all of the members that are exposed by objects of the parent class
- At a less abstract level, an is-a relationship guarantees that in a client a variable of the derived class type could appear anywhere a variable of the parent class type was legal, without causing a type error
- Moreover, the derived class objects should be behaviorally equivalent to the parent class objects
- Ada -> subtype Small_Int is Integer range **-100, ..., 0, ..., 100**;
    - Every Small_Int variable is, in a sense, an Integer variable
    - Variables of Small_Int type have all of the operations of Integer variables but can store only a subset of the values possible in Integer
    - Furthermore, every Small_Int variable can be used anywhere an Integer variable can be used
    - That is, every Small_Int variable is, in a sense, an Integer variable

Miscellaneous

```
//Base class
class Parent
{
   public:
     int id_p;
};

class Child : public Parent
{
   public:
     int id_c;
};

class child_derived: public Child
{

};
```

```
int main()
{
      Parent obj;
      obj.id_p = 9;
      Child obj1;
      obj1.id_c = 7;
      obj1.id_p = 91;
      cout << “Parent id is " <<  obj.id_p <<
endl;
      cout << “Child id is " <<  obj1.id_p <<
endl;

      return 0;
  }
```

Inheritance -> Is Unidirectional

Types

- **Single Inheritance**
- Multiple Inheritance

O/P: Parent id is 9
Child id is 91

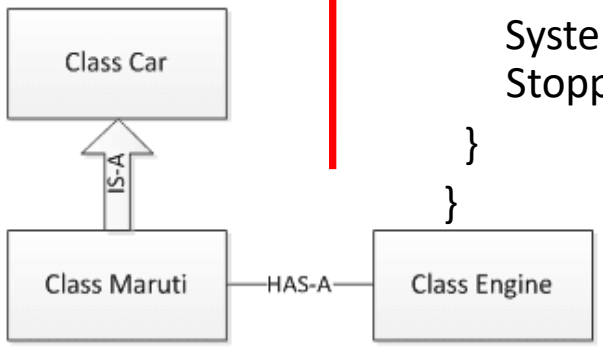
Miscellaneous

Java:

Is-A vs Has-A

```
class Car {
    private String color;
    private int maxSpeed;
    public void carInfo()
    {
       System.out.println("Car Color= "+color
      + " Max Speed= " + maxSpeed);
   }
   public void setColor(String color)
   {
     this.color = color;
   }
   public void setMaxSpeed(int maxSpeed)
   {
     this.maxSpeed = maxSpeed;
   }
}
```

```
class Maruti extends Car
{
    public void MarutiStartDemo()
    {
      Engine MarutiEngine = new Engine();
      MarutiEngine.start();
    }
}
public class Engine
{
   public void start()
   {
      System.out.println("Engine Started:");
   }
   public void stop()
   {
      System.out.println("Engine
      Stopped:");
   }
 }
```



Composition (HAS-A) simply mean the use of instance variables that are references to other objects

Miscellaneous

```
public class RelationsDemo
{
   public static void main(String[] args)
   {
      Maruti myMaruti = new Maruti();
      myMaruti.setColor("RED");
      myMaruti.setMaxSpeed(180);
      myMaruti.carInfo();
      myMaruti.MarutiStartDemo();
    }
}
```

O/P:

```
Car Color= RED Max Speed= 180
Engine Started:
```

Miscellaneous

- In OOP, IS-A relationship is completely inheritance
- This means, that the child class is a type of parent class. Eg: An apple is a fruit. So you will extend fruit to get apple
- Composition means creating instances which have references to other objects
- Eg: A room has a table. So you will create a class room and then in that class create an instance of type table

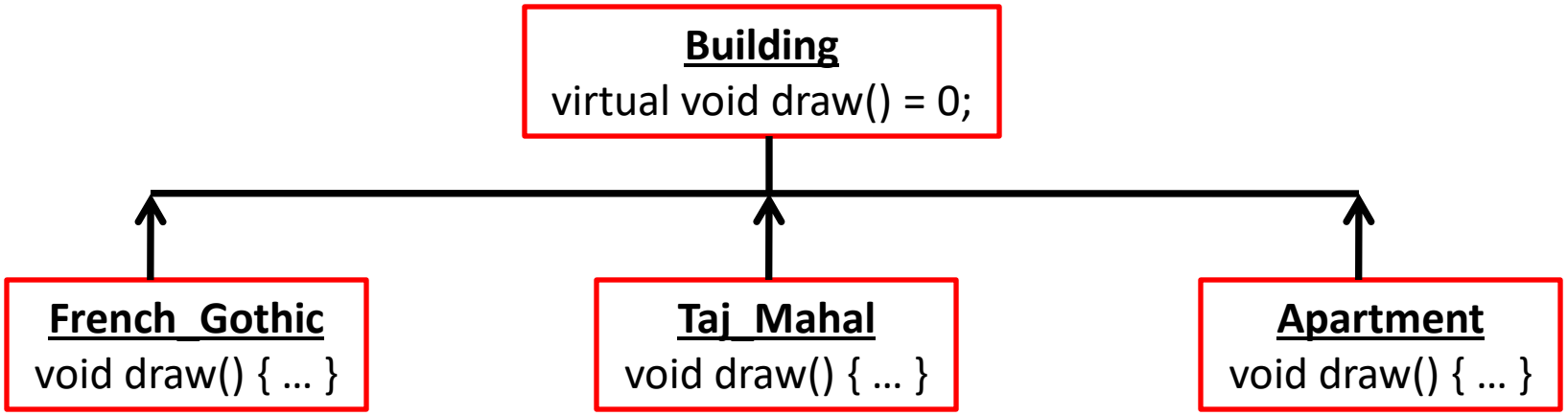
```
class Room {

    Table table = new Table();

}
```

- A HAS-A relationship is dynamic (run time) binding while inheritance is a static (compile time) binding
- If you just want to reuse the code and you know that the two are not of same kind use composition
- Eg: You cannot inherit an oven from a kitchen. A kitchen HAS-A oven.
- When you feel there is a natural relationship like Apple is a Fruit use inheritance

Miscellaneous

Need for Virutal Function:

- Suppose a program defined a Building class and a collection of subclasses for specific types of buildings, for instance, French_Gothic
- It probably would not make sense to have an implemented draw method in Building
- But because all of its descendant classes should have such an implemented method, the protocol (but not the body) of that method is included in Building



Design Issues for Object-Oriented Languages

- There are a wide variety of ways in which a subclass could differ from its base or parent class
  - Subclass could have additional methods
  - Could have fewer methods
  - Types of some of the parameters could be different in one or more methods
  - Return type of some method could be different
  - Number of parameters of some method could be different
  - Body of one or more of the methods could be different
- Most programming languages severely restrict the ways in which a subclass can differ from its base class
- In most cases, the language rules restrict the subclass to be a subtype of its parent class
- A derived class is called a subtype if it has an is-a relationship (derived using public access specifier) with its parent class

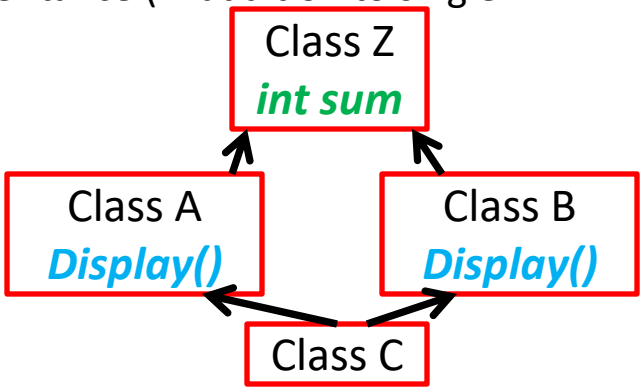
Design Issues for Object-Oriented Languages

- Characteristics of a subclass that ensure that it is a subtype are as follows:
  - Methods of the subclass that override parent class methods must be type compatible with their corresponding overridden methods
    - Compatible here means that a call to an overriding method can replace any call to the overridden method in any appearance in the client program without causing type errors
    - That means that every overriding method must have the same number of parameters as the overridden method and the types of the parameters and the return type must be compatible with those of the parent class
    - Having an identical number of parameters and identical parameter types and return type would, of course, guarantee compliance of a method
- Derivation process for subtypes must require that public entities of the parent class are inherited as public entities in the subclass

Design Issues for Object-Oriented Languages

Single and Multiple Inheritance

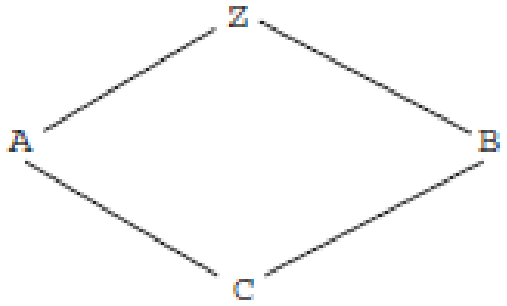
- Another simple issue is: Does the language allow multiple inheritance (in addition to single inheritance)?
- Reasons lie in two categories: Complexity and Efficiency
- Additional complexity is illustrated by several problems



- If C needs to reference both versions of display, how can that be done?
- This ambiguity problem is further complicated when the two parent classes both define identically named methods and one or both of them must be overridden in the subclass
- Another issue arises if both A and B are derived from a common parent Z, and C has both A and B as parent classes -> Diamond or Shared Inheritance
- In this case, both A and B should include Z's inheritable variables
- Suppose Z includes an inheritable variable named "sum"
- The question is whether C should inherit both versions of sum or just one, and if just one, which one?
- There may be programming situations in which just one of the two should be inherited, and others in which both should be inherited

Design Issues for Object-Oriented Languages

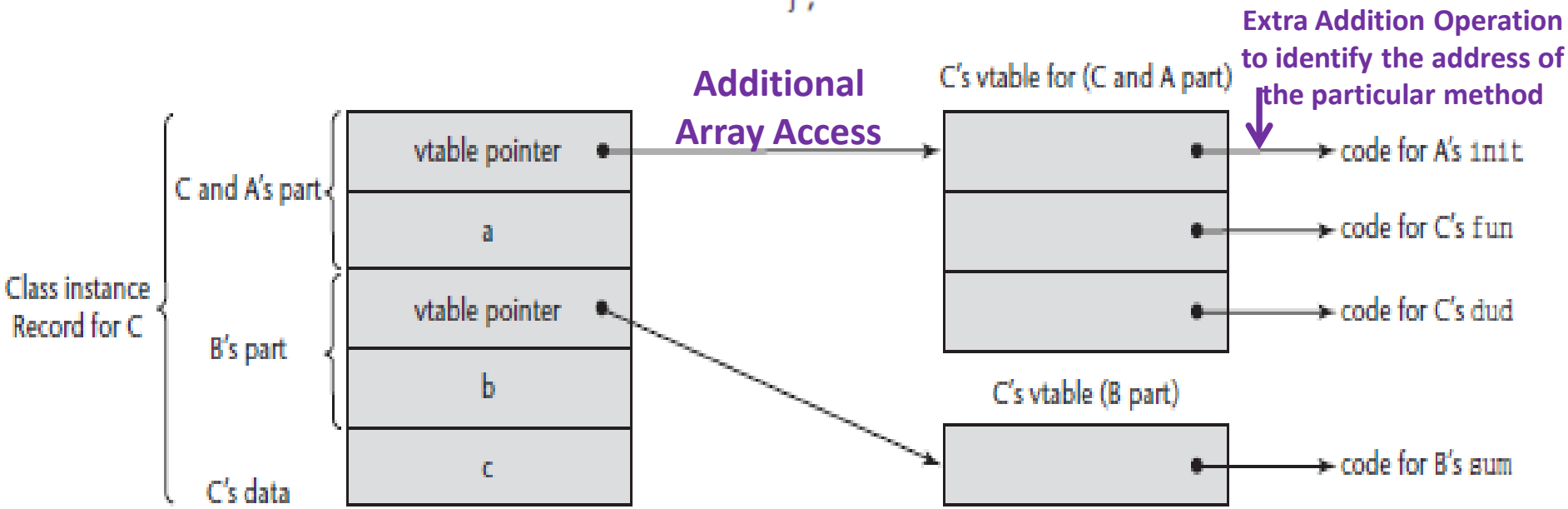
- The question of efficiency may be more perceived than real
- In C++, for example, supporting multiple inheritance requires just one additional array access and one extra addition operation for each dynamically bound method call
- Although this operation is required even if the program does not use multiple inheritance, it is a small additional cost



- Interfaces (abstract class) are an alternative to multiple inheritance
- Interfaces provide some of the benefits of multiple inheritance but have fewer disadvantages

Miscellaneous

```
class A {
  public:
    int a;
    virtual void fun() { . . . }
    virtual void init() { . . . }
};
class B {
  public:
    int b;
    virtual void sum() { . . . }
};
class C : public A, public B {
  public:
    int c;
    virtual void fun() { . . . }
    virtual void dud() { . . . }
};
```



Design Issues for Object-Oriented Languages

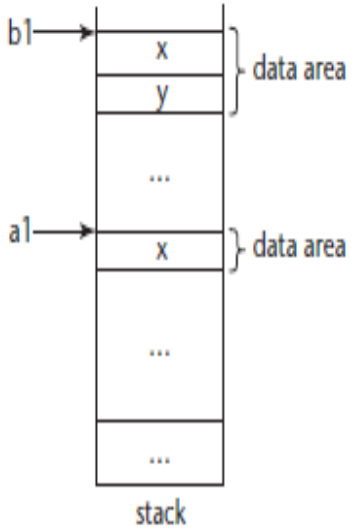
Allocation and Deallocation of Objects

- Two design questions concerning the allocation and deallocation of objects
  - First question is the place from which objects are allocated
    - If they behave like the abstract data types, then perhaps they can be allocated from anywhere
      - ❑ This means they could be allocated from the run-time stack or explicitly created on the heap with an operator or function, such as new
    - If they are all heap dynamic, there is the advantage of having a uniform method of creation and access through pointer or reference variables
      - ❑ This design simplifies the assignment operation for objects, making it in all cases only a pointer or reference value change
      - ❑ Also allows references to objects to be implicitly dereferenced, simplifying the access syntax

Design Issues for Object-Oriented Languages

- If objects are stack dynamic (value variables), there is a problem with regard to subtypes
  - If class B is a child of class A and B is a subtype of A, then an object of B type can be assigned to a variable of A type
  - Eg: If b1 is a variable of B type and a1 is a variable of A type, then a1 = b1; is a legal statement
  - If a1 and b1 are references to heap-dynamic objects, there is no problem—the assignment is a simple pointer assignment
  - However, if a1 and b1 are stack dynamic, then they are value variables and, if assigned the value of the object, must be copied to the space of the target object
  - If B adds a data field to what it inherited from A, then a1 will not have sufficient space on the stack for all of b1
  - The excess will simply be truncated, which could be confusing to programmers who write or use the code
  - This truncation is called object slicing

```
class A {
  int x;
  ...
};
class B : A {
  int y;
  ...
}
```



Design Issues for Object-Oriented Languages

- Second question here is concerned with those cases where objects are allocated from the heap
  - Question is whether deallocation is implicit, explicit, or both
    - ❑ If deallocation is implicit, some implicit method of storage reclamation is required
    - ❑ If deallocation can be explicit, that raises the issue of whether dangling pointers or references can be created

Dynamic and Static Binding

- Dynamic binding of messages to methods is an essential part of object-oriented programming
- Question here is whether all binding of messages to methods is dynamic
- Alternative is to allow the user to specify whether a specific binding is to be dynamic or static
- Advantage of this is that static bindings are faster
- So, if a binding need not be dynamic, why pay the price?

Design Issues for Object-Oriented Languages

Nested Classes

- One of the primary motivations for nesting class definitions is information hiding
- If a new class is needed by only one class, there is no reason to define it so it can be seen by other classes
  - New class can be nested inside the class that uses it
  - In some cases, the new class is nested inside a subprogram, rather than directly in another class
  - The class in which the new class is nested is called the nesting class
- Most obvious design issues associated with class nesting are related to visibility
- Specifically, one issue is: Which of the facilities of the nesting class are visible in the nested class
- The other main issue is the opposite: Which of the facilities of the nested class are visible in the nesting class?

```
Class A                          <-- Nesting Class
{
   public:
      int a = 5;
      void display()
      {
        printf("Show");
      }
      Class B
      {                          <-- Nested Class
        public:
          int b = 10;
          public int calc(int a,
int b)
          {
            return (a + b);
          }
      }
}
```

Design Issues for Object-Oriented Languages

Initialization of Objects

- Initialization issue is whether and how objects are initialized to values when they are created
- First question is whether objects must be initialized manually or through some implicit mechanism
- When an object of a subclass is created, is the associated initialization of the inherited parent class member implicit or must the programmer explicitly deal with it

Support for Object-Oriented Programming in C++

- C++ was the first widely used object-oriented programming language and is still among the most popular
- So, naturally, it is the one with which other languages are often compared

General Characteristics

- To main backward compatibility with C, C++ retains the type system of C and adds classes to it
- C++ has both traditional imperative-language types and the class structure of an object-oriented language
- C++ is a hybrid language, supporting both procedural programming and object- oriented programming
- Objects of C++ can be static, stack dynamic or heap dynamic
- Explicit deallocation using the delete operator is required for heap-dynamic objects because C++ does not include implicit storage reclamation

Support for Object-Oriented Programming in C++

- Many class definitions include a destructor method, which is implicitly called when an object of the class ceases to exist
- Destructor is used to deallocate heap-allocated memory that is referenced by data members
    - It may also be used to record part or all of the state of the object just before it dies, usually for debugging purposes

Inheritance

- A C++ class can be derived from an existing class, which is then its parent, or base, class
- C++ class can also be stand-alone, without a superclass
- Data defined in a class definition -> Data Members of that class
- Functions defined in a class definition -> Member Functions of that class (Methods)
- Some or all of the members of the base class may be inherited by the derived class, which can also add new members and modify inherited member functions

Support for Object-Oriented Programming in C++

- All C++ objects must be initialized before they are used
- Therefore, all C++ classes include at least one constructor method that initializes the data members of the new object
- Constructor methods are implicitly called when an object is created
- If any of the data members are pointers to heap-allocated data, the constructor allocates that storage. Eg: int *a[] = new array[20];
- If a class has a parent, the inherited data members must be initialized when the subclass object is created
    - To do this, the parent constructor is implicitly called
- When initialization data must be furnished to the parent constructor, it is given in the call to the subclass object constructor

```
subclass (subclass parameters) : parent_class (superclass parameters) {
. . .
}
```

Miscellaneous

```
class Base
{
      int x;
   public:
      Base(int i)
      {
         x = i;
         cout << "Base Parameterized Constructor\n";
         cout << i;
      }
};
class Derived : public Base
{
      int y;
   public:
      Derived(int j, int p):Base(p)
      {
        y = j;
        cout << "\nDerived Parameterized Constructor\n";
        cout << y;
      }
};
```

Receives 2 parameters as argument and passes one parameter to base class constructor

```
int main()
{
    Derived d(10, 5) ;
}
```

O/P: Base Parameterized Constructor
5
Derived Parameterized Constructor
10

Miscellaneous

```
class Base
{
      int x;
   public:
      Base(int i)
      {
         x = i;
         cout << "Base Parameterized Constructor\n";
         cout << i;
      }
};
class Derived : public Base
{
      int y;
   public:
      Derived(int j):Base(j)
      {
        y = j;
        cout << "\nDerived Parameterized Constructor\n";
        cout << y;
      }
};
```

Receives 1 parameter as argument and passes that parameter to base class constructor

```
int main()
{
    Derived d(10) ;
}
```

O/P: Base Parameterized Constructor
10
Derived Parameterized Constructor
10

Support for Object-Oriented Programming

- If no constructor is included in a class definition, the compiler includes a trivial constructor
- This default constructor calls the constructor of the parent class, if there is a parent class

```
class derived_class_name : derivation_mode base_class_name
 {data member and member function declarations} ;
```

```
class base_class {
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};

class subclass_1 : public base_class {...};
class subclass_2 : private base_class {...};
```

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

- In subclass_1, b and y are protected, and c and z are public
- In subclass_2, b, y, c, and z are private
- No derived class of subclass_2 can have members with access to any member of base_class
- The data members a and x in base_class are not accessible in either subclass_1 or subclass_2

Support for Object-Oriented Programming

- Note that private-derived subclasses cannot be subtypes
- Eg: If the base class has a public data member, under private derivation that data member would be private in the subclass
- Therefore, if an object of the subclass were substituted for an object of the base class, accesses to that data member would be illegal on the subclass object
- The is-a relationship would be broken
- Under private class derivation, no member of the parent class is implicitly visible to the instances of the derived class
- Any member that must be made visible must be reexported in the derived class
- This reexportation in effect exempts a member from being hidden even though the derivation was private

```
class subclass_3 : private base_class {
  base_class :: c;
  ...
}
```

Now, instances of subclass_3 can access c. As far as c is concerned, it is as if the derivation had been public. The double colon (::) in this class definition is a scope resolution operator. It specifies the class where its following entity is defined.

Miscellaneous

```
//Base class
class Parent
{
   public:
     int id_p;
};

class Child : private Parent
{
   public:
     int id_c;
};

class child_derived: public Child
{

};
```

```
int main()
{
      Parent obj;
      obj.id_p = 9;
      Child obj1;
      obj1.id_c = 7;
      obj1.id_p = 91;
      cout << "Child id is " <<  obj.id_p <<
endl;
      cout << "Parent id is " <<  obj1.id_p <<
endl;

      return 0;
  }
```

Inheritance -> Is Unidirectional

Types

- **Single Inheritance**
- Multiple Inheritance

O/P: Error in line: obj1.id_p = 91;

Miscellaneous

```
#include <iostream>
using namespace std;
//Base class
class Parent
{
   public:
     int id_p;
};

class Child : private Parent
{
   public:
     int id_c;
     int Parent :: id_p;
};

class child_derived: public Child
{

};
```

```
int main()
{
      Parent obj;
      obj.id_p = 9;
      Child obj1;
      obj1.id_c = 7;
      obj1.id_p = 91;
      cout << "Child id is " <<  obj.id_p <<
endl;
      cout << "Parent id is " <<  obj1.id_p <<
endl;

      return 0;
  }
```

Inheritance -> Is Unidirectional

Types

- **Single Inheritance**
- Multiple Inheritance

O/P: Error in line: obj1.id_p = 91;

Support for Object-Oriented Programming

An example for the purpose and use of private derivation

```
class single_linked_list {
  private:
    class node {
      public:
        node *link;
        int contents;
    };
    node *head;
  public:
    single_linked_list() {head = 0};
    void insert_at_head(int);
    void insert_at_tail(int);
    int remove_at_head();
    int empty();
};
```

Nesting Class

Nested Class

```
class stack : public single_linked_list {
  public:
    stack() {}
    void push(int value) {
      insert_at_head(value);
    }
    int pop() {
      return remove_at_head();
    }
};
class queue : public single_linked_list {
  public:
    queue() {}
    void enqueue(int value) {
      insert_at_tail(value);
    }
    int dequeue() {
      remove_at_head();
    }
};
```

- Note that nested classes have no special access to members of the nesting class. Only static data members of the nesting class are visible to methods of the nested class
- Enclosing class, single_linked_list, has just a single data member, a pointer to act as the list's header. It contains a constructor function, which simply sets head to the null pointer value.

Support for Object-Oriented Programming

- A client of a stack object could call insert_at_tail, thereby destroying the integrity of its stack
- Likewise, a client of a queue object could call insert_at_head
- These unwanted accesses are allowed because both stack and queue are subtypes of single_linked_list
- Our two example derived classes can be written to make them not subtypes of their parent class by using private, rather than public, derivation
- Then, both will also need to reexport empty, because it will become hidden to their instances
- This situation illustrates the motivation for the private-derivation option

Support for Object-Oriented Programming

```
class stack_2 : private single_linked_list {
  public:
    stack_2() {}
    void push(int value) {
      single_linked_list :: insert_at_head(value);
    }
    int pop() {
      return single_linked_list :: remove_at_head();
    }
    single_linked_list:: empty();
};
class queue_2 : private single_linked_list {
  public:
    queue_2() {}
    void enqueue(int value) {
      single_linked_list :: insert_at_tail(value);
    }
    int dequeue() {
      single_linked_list :: remove_at_head();
    }
    single_linked_list:: empty();
};
```

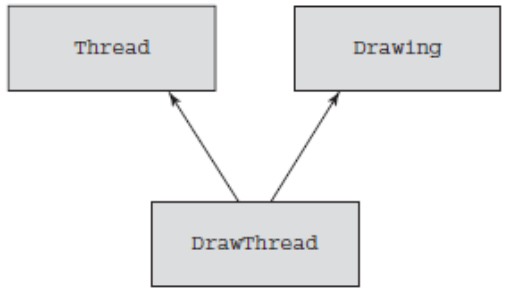
Support for Object-Oriented Programming

- The two versions of stack and queue illustrate the difference between subtypes and derived types that are not subtypes
- The linked list is a generalization of both stacks and queues, because both can be implemented as linked lists
- So, it is natural to inherit from a linked-list class to define stack and queue classes
- However, neither is a subtype of the linked-list class, because both make the public members of the parent class private, which makes them inaccessible to clients
- One of the reasons friends are necessary is that sometimes a subprogram must be written that can access the members of two different classes
- Eg: Suppose a program uses a class for vectors and one for matrices, and a subprogram is needed to multiply a vector object times a matrix object
- In C++, the multiply function can be made a friend of both classes

Support for Object-Oriented Programming

- C++ provides multiple inheritance, which allows more than one class to be named as the parent of a new class

```
class Thread { ... };
class Drawing { ... };
class DrawThread : public Thread, public Drawing { ... };
```



- Class DrawThread inherits all of the members of both Thread and Drawing
- If both Thread and Drawing happen to include members with the same name, they can be unambiguously referenced in objects of class DrawThread by using the scope resolution operator (::)
- Overriding methods in C++ must have exactly the same parameter profile as the overridden method
- If there is any difference in the parameter profiles, the method in the subclass is considered a new method that is unrelated to the method with the same name in the ancestor class
- The return type of the overriding method ~~either~~ must be the same as that of the overridden method ~~or must be a publicly derived type of the return type of the overridden method~~

Miscellaneous

```
class A
{
    int idA;
    void setId(int i)
    {
      idA = i;
    }
    int getId()
    {
     return idA;
    }
};
```

```
class B
{
    int idB;
    void setId(int i)
    {
      idB = i;
    }
    int getId()
    {
      return idB;
    }
};
class AB : public A, public B
{
};
int main()
{
    AB *ab = new AB();
    ab -> setID(5);
}
```

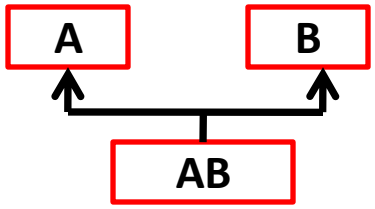
Problem in Multiple Inheritance

O/P: Error: request for member 'setId' is ambiguous
Note: candidates are: void B::setId(int)
Note: void A::setId(int)

Miscellaneous

```
class A
{
   public:
    int idA;
    void setId(int i)
    {
      idA = i;
    }
    int getId()
    {
     return idA;
    }
};
```

```
class B
{
  public:
    int idB;
    void setId(int i)
    {
      idB = i;
    }
    int getId()
    {
      return idB;
    }
};
class AB : public A, public B
{
   public:
    using A::setId;
    using A::getId;
};
int main()
{
   AB *ab = new AB();
   ab->B::setId(10);
}
```



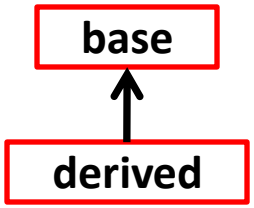
Note: Either use the one defined in "main" function (::) or use the one defined in class AB (using)

Miscellaneous

```
class base
{
public:
   virtual void print ()
   {
     cout<< "print base class" <<endl;
   }

   void show ()
   {
     cout<< "show base class" <<endl;
   }
};

class derived: public base
{
public:
   void print () //print () is already virtual function in derived class, we could also declared as virtual void print () explicitly
   {
     cout<< "print derived class" <<endl;
   }

   void show ()
   {
     cout<< "show derived class" <<endl;
   }
};
```

O/P: print derived class
show base class

```
int main()
{
   base *bptr;
   derived d;
   bptr = &d;

   //virtual function, binded at runtime (Runtime polymorphism)
   bptr->print();

   // Non-virtual function, binded at compile time
   bptr->show();

   return 0;
}
```



This type of polymorphism is achieved by Function Overriding. Function overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden

Types

- Functional Overloading (Compile Time)
- Operator Overloading (Compile Time)
- **Virtual Functions (Run Time)**

Dynamic Binding

- All of the member functions we have defined thus far are statically bound; that is, a call to one of them is statically bound to a function definition
- A C++ object could be manipulated through a value variable, rather than a pointer or a reference (Such an object would be static or stack dynamic)
- However, in that case, the object's type is known and static, so dynamic binding is not needed
- On the other hand, a pointer variable that has the type of a base class can be used to point to any heap-dynamic objects of any class publicly derived from that base class, making it a polymorphic variable
- Publicly derived subclasses are subtypes if none of the members of the base class are private
- Privately derived subclasses are never subtypes
- A pointer to a base class cannot be used to reference a method in a subclass that is not a subtype

Miscellaneous

```
class Vehicle
{
   public:
    Vehicle()
    {
       cout << "This is a Vehicle" << endl;
    }
    virtual void draw()
    {
        cout<<"Vehicle Draw";
    }
};
```

```
class FourWheeler: public Vehicle
{
    public:
      FourWheeler()
      {
          cout << "This is a 4 wheeler
     Vehicle" << endl;
      }
      void draw()
      {
        cout<<"Fourwheeler Draw";
      }
};
void main()
{
    Vehicle *obj = new FourWheeler();
    obj->draw();
}
```

FourWheeler *obj = new Vehicle();

O/P: error: invalid conversion from 'Vehicle*' to 'FourWheeler*' [-fpermissive]

O/P: This is a Vehicle
This is a 4 wheeler Vehicle
Fourwheeler Draw

Dynamic Binding

- C++ does not allow value variables (as opposed to pointers or reference) to be polymorphic
- When a polymorphic variable is used to call a member function overridden in one of the derived classes, the call must be dynamically bound to the correct member function definition
- Member functions that must be dynamically bound must be declared to be virtual functions by preceding their headers with the reserved word virtual, which can appear only in a class body

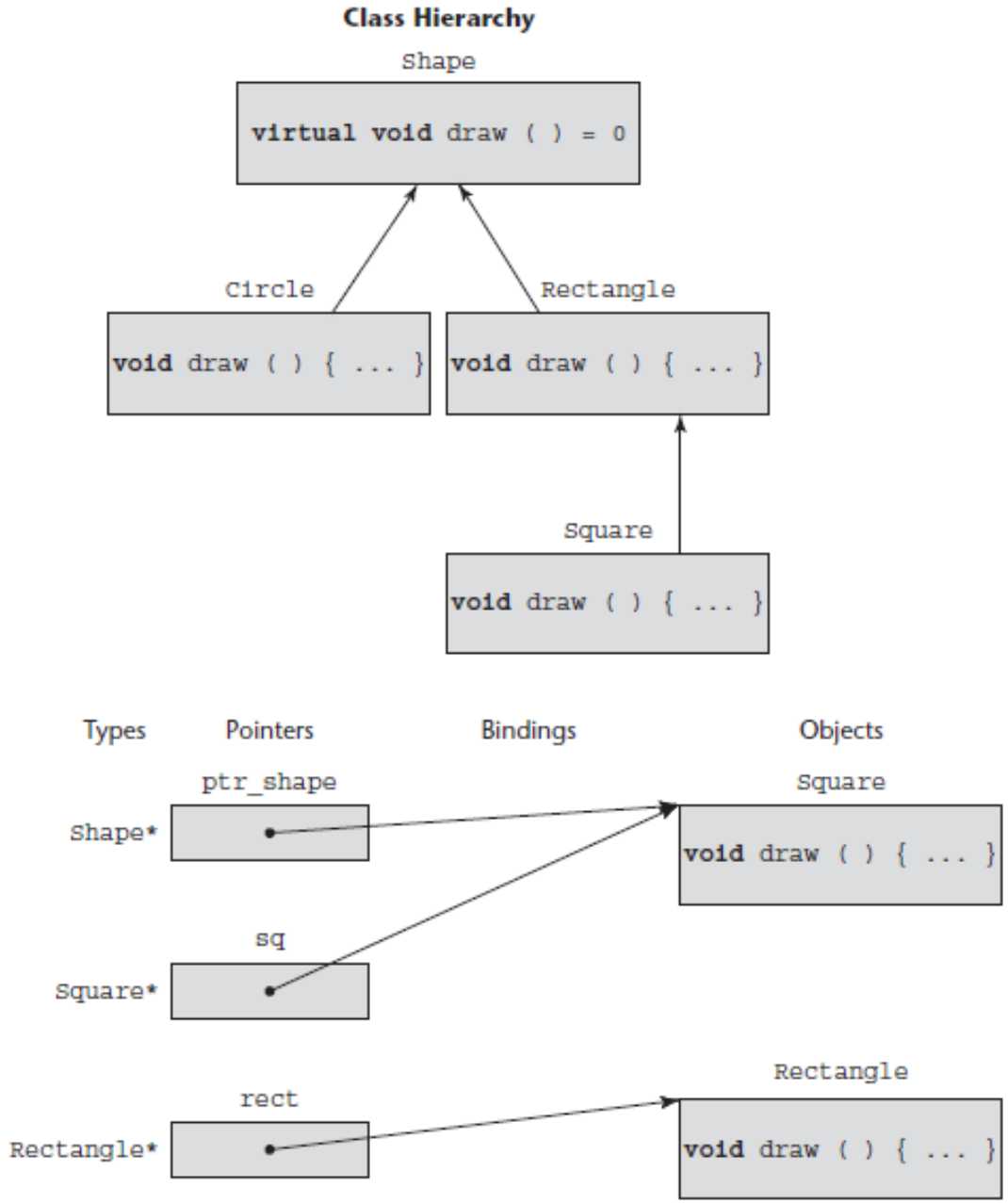
```
class Shape {
  public:
    virtual void draw() = 0;
  ...
};
class Circle : public Shape {
  public:
    void draw() { ... }
  ...
};
class Rectangle : public Shape {
  public:
    void draw() { ... }
  ...
};
class Square : public Rectangle {
  public:
    void draw() { ... }
  ...
};
```

- Rect -> Value Variable
- Ptr_shape -> Polymorphic Variable

```
Square* sq = new Square;
Rectangle* rect = new Rectangle;
Shape* ptr_shape;

ptr_shape = sq;          // Now ptr_shape points to a
                         //  Square object
ptr_shape->draw();       // Dynamically bound to the draw
                         //  in the Square class
rect->draw();            // Statically bound to the draw
                         //  in the Rectangle class
```

Versions of draw must be defined to be virtual. When a call to draw is made with a pointer to the base class of the derived classes, that call must be dynamically bound to the member function of the correct derived class.

Dynamic Binding



Dynamic Binding

- Notice that the draw function in the definition of the base class shape is set to 0
- This peculiar syntax is used to indicate that this member function is a pure virtual function, meaning that it has no body and it cannot be called
- It must be redefined in derived classes if they call the function
- The purpose of a pure virtual function is to provide the interface of a function without giving any of its implementation
- Pure virtual functions are usually defined when an actual member function in the base class would not be useful
- Any class that includes a pure virtual function is an abstract class
- An abstract class can include completely defined methods
- It is illegal to instantiate an abstract class
- If a subclass of an abstract class does not redefine a pure virtual function of its parent class, that function remains as a pure virtual function in the subclass and the subclass is also an abstract class
- Abstract classes and inheritance together support a powerful technique for software development

Dynamic Binding

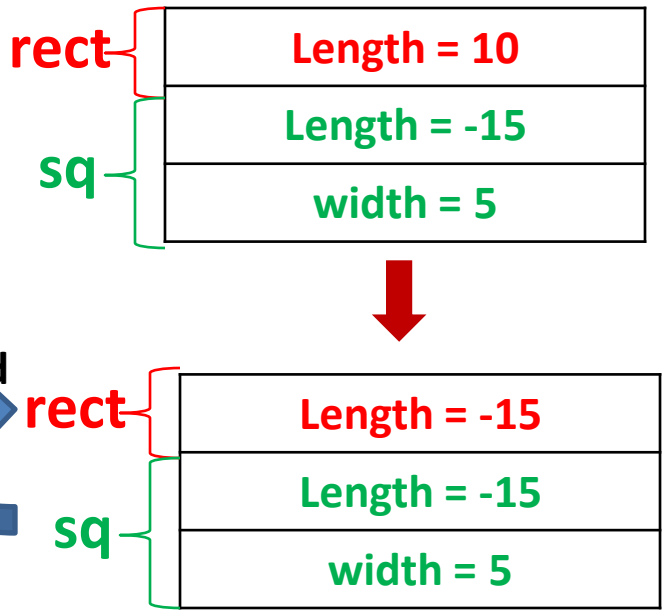
- Dynamic binding allows the code that uses members like draw to be written before all or even any of the versions of draw are written
- New derived classes could be added years later, without requiring any change to the code that uses such dynamically bound members
- This is a highly useful feature of object-oriented languages
- Reference assignments for stack-dynamic objects are different from pointer assignments for heap-dynamic objects

```
Square sq;          // Allocate a Square object on the stack
Rectangle rect;     // Allocate a Rectangle object on
                    //  the stack
rect = sq;          // Copies the data member values from
                    //  the Square object
rect.draw();        // Calls the draw from the Rectangle
                    //  object
```

- In the assignment rect = sq, the member data from the object referenced by sq would be assigned to the data members of the object referenced by rect, but rect would still reference the Rectangle object
- Therefore, the call to draw through the object referenced by rect would be that of the Rectangle class
- If rect and sq were pointers to heap-dynamic objects, the same assignment would be a pointer assignment, which would make rect point to the Square object, and a call to draw through rect would be bound dynamically to the draw in the Square object

Miscellaneous

```
class Rectangle
{
   int Length;
   void draw() { .... }
};
class Square : public Rectangle
{
   int width;
   void draw() { .... }
};
int main()
{
     Square sq;
     sq.Length = -15;
     cout << sq.Length << endl;
     sq.width = 5;
     cout << sq.width << endl;
     Rectangle rect;
     rect.Length = 10;
     cout << rect.Length << endl;
     rect = sq;
     cout << rect.Length << endl;
     rect.draw()
}
```



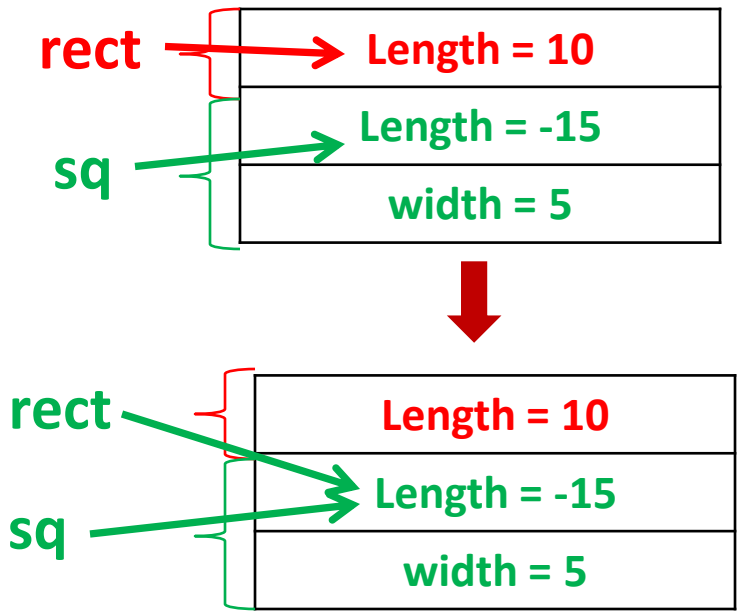
O/P: -15
5
10
-15

It will call the method "draw" in class "**Rectangle**"

Note: In the assignment rect = sq, the member data from the object referenced by sq would be assigned to the data members of the object referenced by rect, but rect would still reference the Rectangle object

Miscellaneous

```
class Rectangle
{
   int Length;
   void draw() { .... }
};
class Square : public Rectangle
{
   int width;
   void draw() { .... }
};
int main()
{
     Square *sq;
     sq.Length = -15;
     sq.width = 5;
     Rectangle *rect;
     rect.Length = 10;
     rect = sq;
     rect.draw()
}
```



Note: If rect and sq were pointers to heap-dynamic objects, the same assignment would be a pointer assignment, which would make rect point to the Square object, and a call to draw through rect would be bound dynamically to the draw in the Square object

It will call the method "draw" in class "**Square**"

Implementation of Object-Oriented Constructs

- Two parts of language support for object-oriented programming that pose interesting questions for language implementers -> Storage structures for instance variables; Dynamic bindings of messages to methods

Instance Data Storage

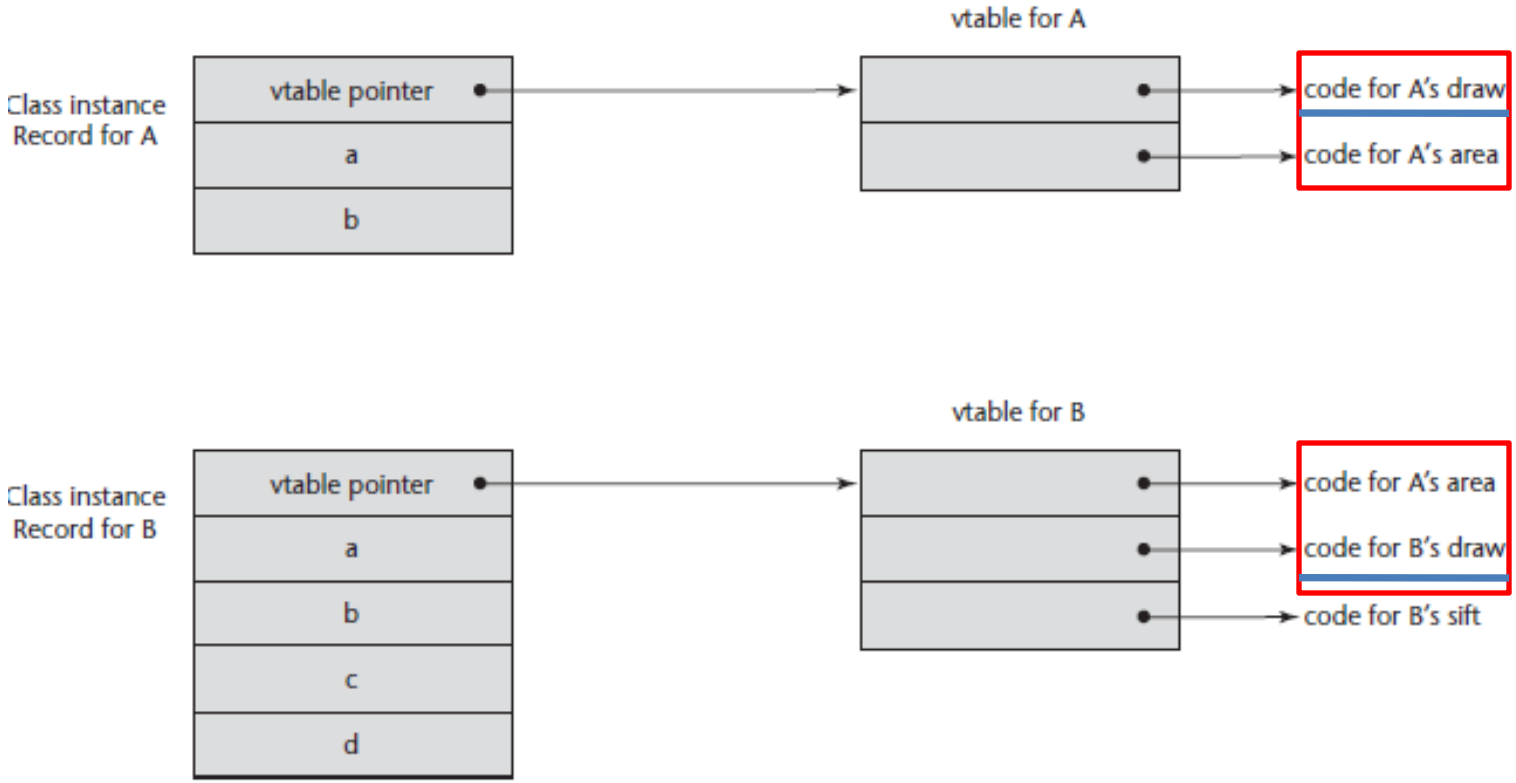
- In C++, classes are defined as extensions of C's record structures—structs
- Similarity suggests a storage structure for the instance variables of class instances—that of a record -> Class Instance Record (CIR)
- Structure of a CIR is static, so it is built at compile time and used as a template for the creation of the data of class instances
- Every class has its own CIR
- When a derivation takes place, the CIR for the subclass is a copy of that of the parent class, with entries for the new instance variables added at the end
- Because the structure of the CIR is static, access to all instance variables can be done as it is in records, using constant offsets from the beginning of the CIR instance
- This makes these accesses as efficient as those for the fields of records

Implementation of Object-Oriented Constructs

Java:

```
public class A {
  public int a, b;
  public void draw() { ... }
  public int area() { ... }
}
```

```
public class B extends A {
  public int c, d;
  public void draw() { ... }
  public void sift() { ... }
}
```



Dynamic Binding of Method Calls to Methods

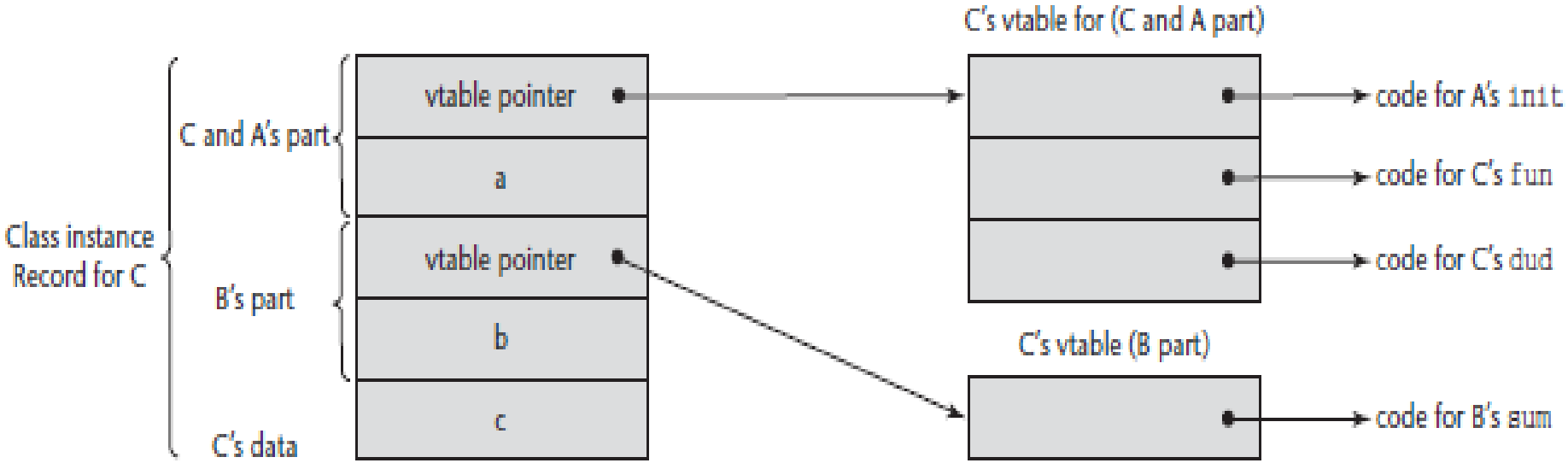
- Methods in a class that are statically bound need not be involved in the CIR for the class
- However, methods that will be dynamically bound must have entries in this structure
- Such entries could simply have a pointer to the code of the method, which must be set at object creation time
- Calls to a method could then be connected to the corresponding code through this pointer in the CIR
- Drawback to this technique is that every instance would need to store pointers to all dynamically bound methods that could be called from the instance

Dynamic Binding of Method Calls to Methods

- Notice that the list of dynamically bound methods that can be called from an instance of a class is the same for all instances of that class
- Therefore, the list of such methods must be stored only once
- So the CIR for an instance needs only a single pointer to that list to enable it to find called methods
- The storage structure for the list is often called a virtual method table (vtable)
- Method calls can be represented as offsets from the beginning of the vtable
- Polymorphic variables of an ancestor class always reference the CIR of the correct type object, so getting to the correct version of a dynamically bound method is assured
- Multiple inheritance complicates the implementation of dynamic binding

Dynamic Binding of Method Calls to Methods

```
class A {
  public:
    int a;
    virtual void fun() { . . . }
    virtual void init() { . . . }
};
class B {
  public:
    int b;
    virtual void sum() { . . . }
};
class C : public A, public B {
  public:
    int c;
    virtual void fun() { . . . }
    virtual void dud() { . . . }
};
```



Dynamic Binding of Method Calls to Methods

- C class inherits the variable a and the init method from the A class
- It redefines the fun method, although both its fun and that of the parent class A are potentially visible through a polymorphic variable (of type A)
- From B, C inherits the variable b and the sum method
- C defines its own variable, c, and defines an uninherited method, dud
- A CIR for C must include A's data, B's data, and C's data, as well as some means of accessing all visible methods
- Under single inheritance, the CIR would include a pointer to a vtable that has the addresses of the code of all visible methods
- With multiple inheritance, however, it is not that simple
- There must be at least two different views available in the CIR—one for each of the parent classes, one of which includes the view for the subclass, C
- This inclusion of the view of the subclass in the parent class's view is just as in the implementation of single inheritance
- There must also be two vtables: one for the A and C view and one for the B view
- The first part of the CIR for C in this case can be the C and A view, which begins with a vtable pointer for the methods of C and those inherited from A, and includes the data inherited from A
- Following this in C's CIR is the B view part, which begins with a vtable pointer for the virtual methods of B, which is followed by the data inherited from B and the data defined in C