University of Southern California

Viterbi School of Engineering

EE599 – Special Topics: Software Design for Electrical Engineers

Basics of Object-Oriented Programming

Reference: Online resources (articles, papers, etc.)

Procedural Programming

- With a procedural approach, first we should concentrate on the procedures and then think about how to represent the data
- Program = Algorithm + Data Structure
- Data and operations on the data are separated

- Procedures are often hard to reuse
- Programs are often hard to extend and maintain

Objected-Oriented Programming (OOP)

- Focus on creating objects rather than procedures/functions
- Objects have both data and procedures that manipulate that data
- Data in an object are known as *attributes*
- Procedures/functions in an object are known as *methods*

- Objects of the same class have the same data elements and methods
- Save development time by reusing code

Procedural Programming vs OOP

- We use the following example to compare the two programming styles
- Example: Basic stock trading
  - Store the following information: name of company, number of stocks owned, value of each share
  - For each stock, customer can buy, sell and update

Procedural Programming vs OOP (cont.)

- **Procedural Programming:**

Data Structure: Stock

```
Struct Stock{
        char company[30];
        int shares;
        double share_val;
        }
```

Procedure: Buy() {…}

Procedure: Sell() {…}

Procedure: Update() {…}

Procedural Programming vs OOP (cont.)

- **OOP:**
  - **Combine the stocks (data) with the operations on the stocks into objects**
- **A new kind of data type: Stock Class**

```
Class Stock {
  private:
        char company[30];
        int shares;
        double share_val;
  public:
    void Buy (int num, double price) {...}
    void Sell (int num, double price) {…}
    void Update (double price) {…}
};
```

Important Components of OOP

- Object
- Class 
- Encapsulation
- Inheritance
- Polymorphism
- Virtual function
- Abstract Class & Pure Virtual Function
- Package

Object

- **Real-world objects share two characteristics: They all have *states* (attributes or data members) and *behavior* (function members or methods or operations)**

	State	behavior
Desk	On, off	Turn on, turn off
Desktop radio	On, off, current volume, current station	Turn on, turn off, increase volume, decrease volume, seek, scan and tune
Bicycle	Current gear, current pedal cadence, current speed	Changing gear, changing pedal cadence, applying brakes

- **State is represented by attributes (fields in some programming languages)**
- **Behavior is represented by methods (functions in some programming languages)**

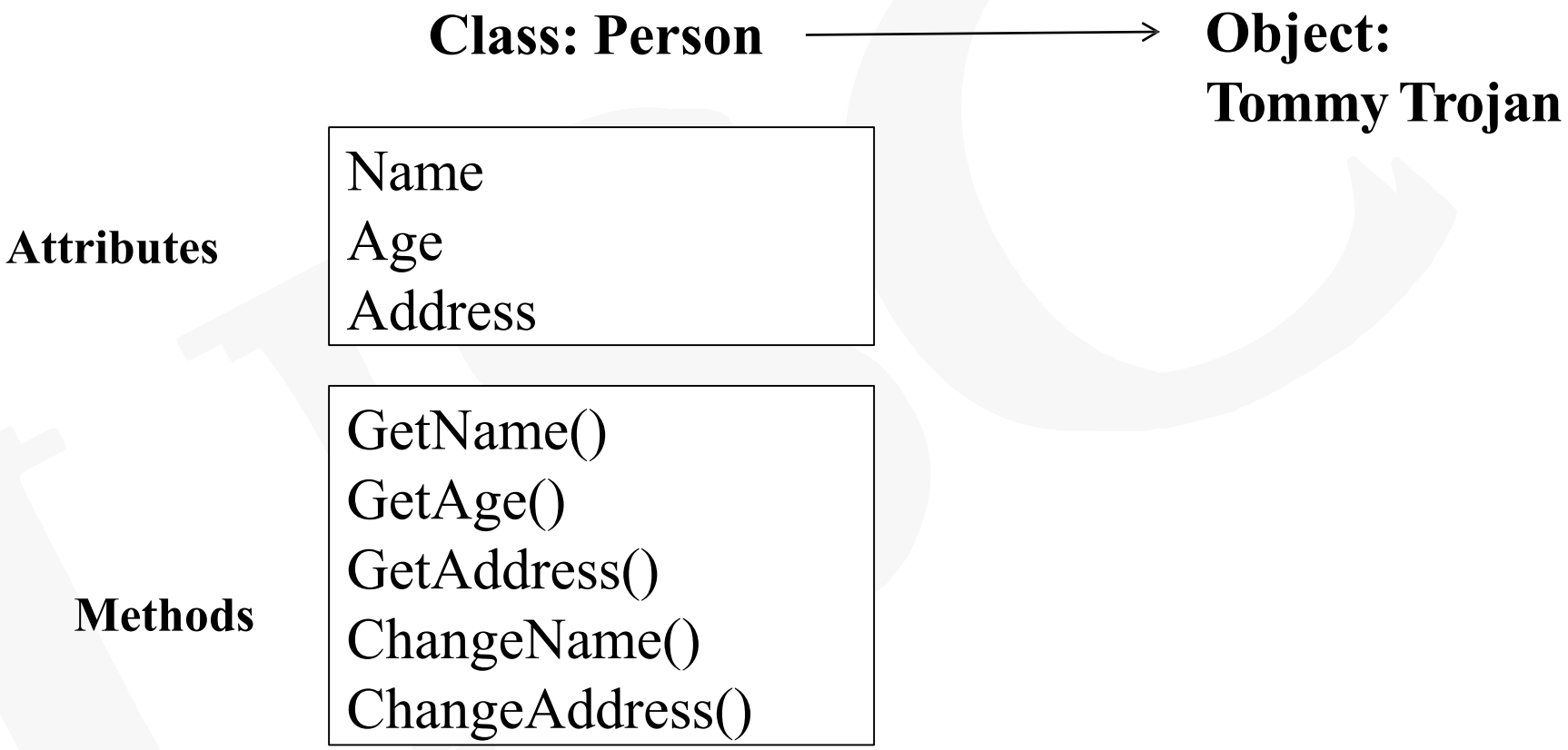
Class

- A class is a description of a set of objects that share the same *state* and *behavior*
  - It defines a "blueprint" for an object
  - It is composed of three things: a name, attributes, and operations

- Note: An object is an instance of a particular class
  - It is built from the "blueprint"

Class – Example I

- **Person**



Class – Example II

- The following **Bicycle** class is one possible implementation of a bicycle:

```
class Bicycle
{   int cadence = 0;  int speed = 0;  int gear = 1;

    void changeCadence(int newValue)
     { cadence = newValue; }

    void changeGear(int newValue)
     { gear = newValue; }

    void speedUp(int increment)
     { speed = speed + increment; }

     void applyBrakes(int decrement)
     { speed = speed - decrement; }

     void printStates()
      { System.out.println("cadence:" +
            cadence + "speed:" +  speed + "gear:" + gear); } }
```

- This code is just a pseudo-code for us to understand the design of class (bicycles as example)
- The fields *cadence*, *speed*, and *gear* represent the object's state
- The methods (*changeCadence*, *changeGear*, *speedUp* etc.) define its interaction with the outside world

Class – Example II (cont.)

- Here we create two separate Bicycle objects and invokes their methods:

```
Bicycle bike1 = new Bicycle();  // Create two different Bicycle objects
Bicycle bike2 = new Bicycle();

bike1.changeCadence(50);  // Invoke methods on those objects
bike1.speedUp(10);
bike1.changeGear(2);
bike1.printStates();

bike2.changeCadence(50);
bike2.speedUp(10);
bike2.changeGear(2);
bike2.changeCadence(40);
bike2.speedUp(10);
bike2.changeGear(3);
bike2.printStates();
```

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

Class – Example III

```
// classes example
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int
y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

Encapsulation

- Gathering the implementation details together and separating them from the abstraction is called *encapsulation*

- Encapsulation means the internal representation of an object is generally hidden from outside the object's definition
- In OOP, objects interact with each other by *messages*. The only thing that an object knows about another object is the object's *interface*. Each object's data and logic are hidden from other objects
  - The interface consists of the methods provided by whoever wrote the class
  - The interface enables the programmer to write code that interacts with class objects, and thus it enables the program to use the class objects

Encapsulation (cont.)

- A class design attempts to separate the public interface from the specifics of the implementation
- The public interface represents the abstraction component of the design
- The user of an object can view the object as a *black box* that provides services
- Instance variables and methods can be added, deleted, or changed, but as long as the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten

Typical encapsulation plan: make class data members private

Encapsulation – Example I

What's your address?

Tommy_Trojan.GetAddress()

Client

Interface

GetName()
GetAge()
GetAddress()
ChangeName()
ChangeAddress()

Name
Age
Address

can't be accessed by client

GetName()
GetAge()
GetAddress()
ChangeName()
ChangeAddress()

Encapsulation – Example II


```
#include <iostream>
using namespace std;
class Adder{
   public:
      // constructor
      Adder(int i = 0) {
         total = i;
      }
      // interface to outside world
      void addNum(int number) {
         total += number;
      }
     // interface to outside world
      int getTotal() {
         return total;
      };
private:
      // hidden data from outside world
      int total;
};

int main( ) {
   Adder a;
   a.addNum(10);
   a.addNum(20);
   a.addNum(30);
   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

Question: Explain how encapsulation worked.

© Shahin Nazarian, All rights reserved

http://www.tutorialspoint.com/cplusplus/cpp_data_encapsulation.htm

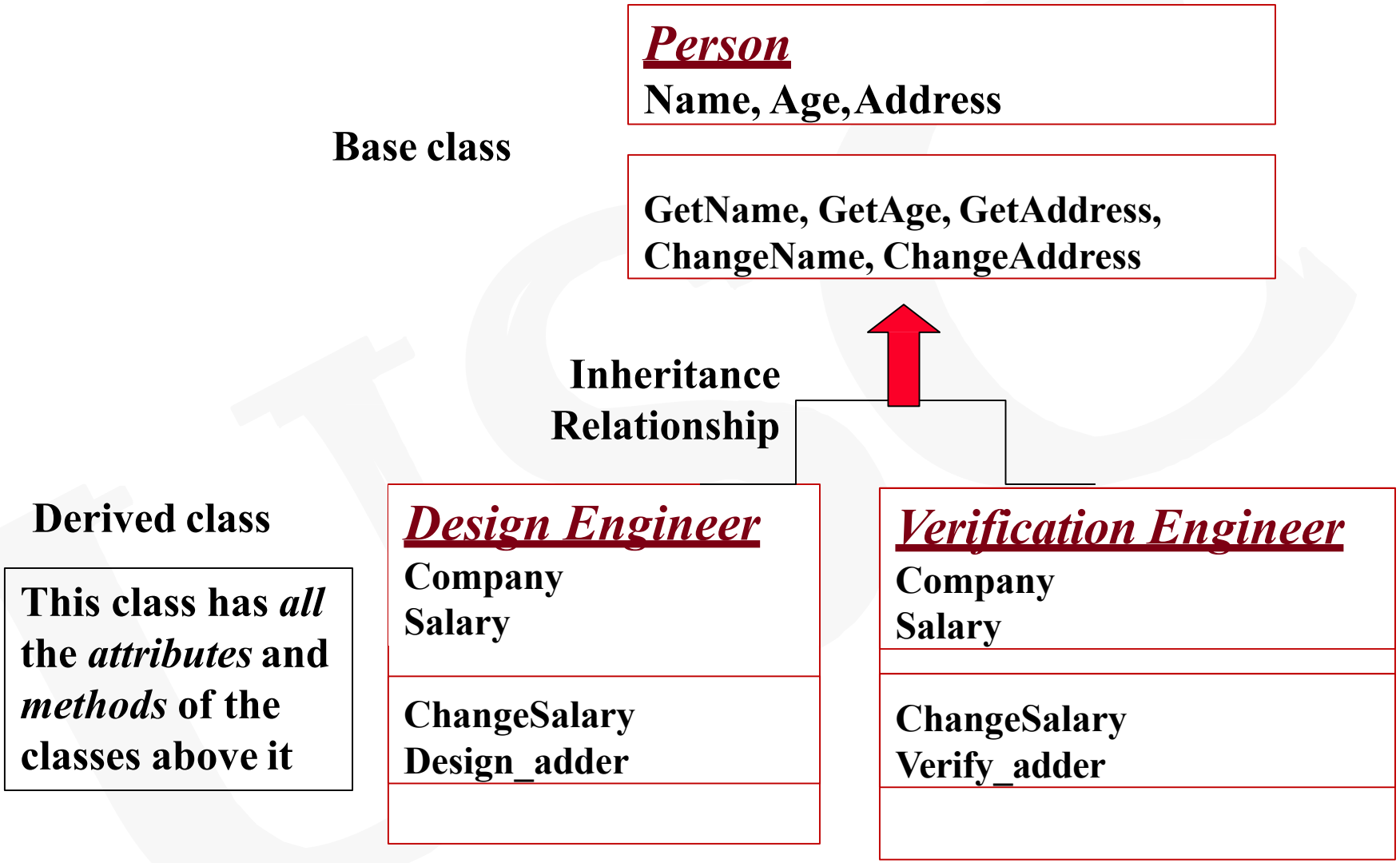
Inheritance

- **Inheritance enables new (derived) classes to receive or *inherit* the properties (data members) and methods (member functions) of existing (base) classes**
  - A class that is used as the basis for inheritance is called a *base* class or *superclass*
  - A class that inherits from a superclass is called a *derived class* or *child* or *subclass*
- **Subclasses and superclasses can be understood in terms of "is a" relationship**
- **A subclass is a more specific instance of a superclass**

Inheritance (cont.)

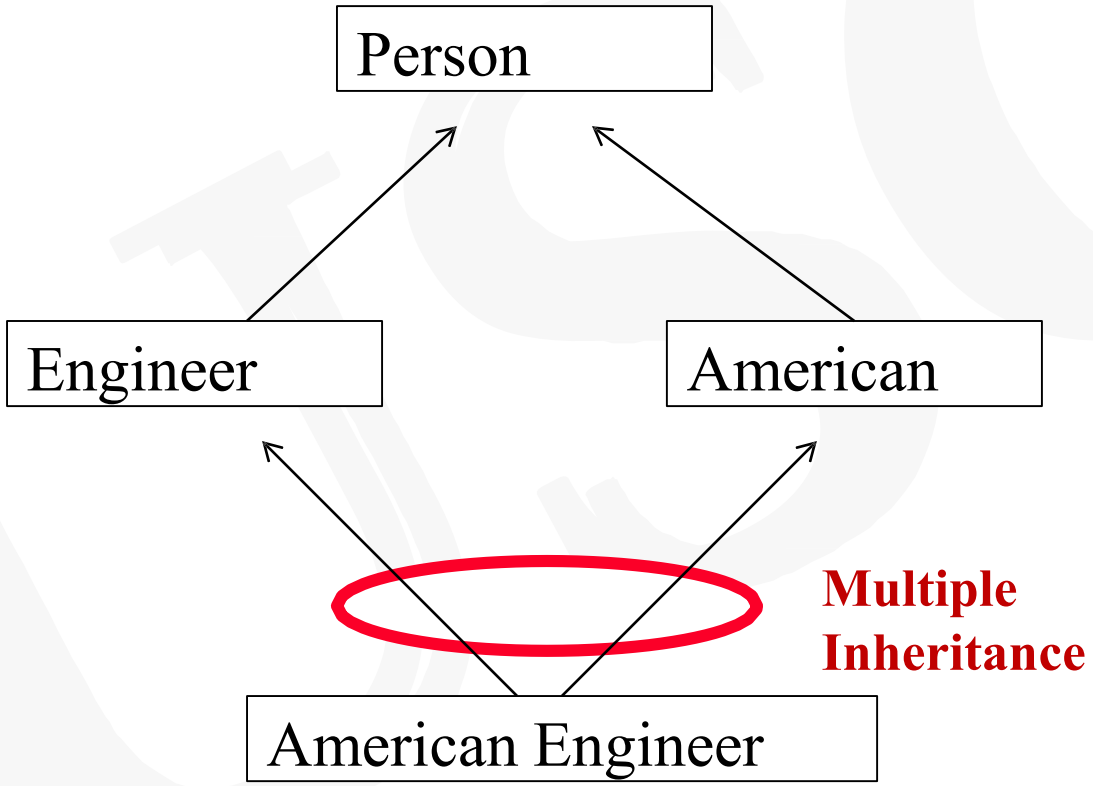
- Here are some things you can do with inheritance:
  - Add functionality to an existing class
  - Add to the data that a class represents
  - Modify how a class method behaves
- Classes with properties in common can be grouped so that their common properties are only defined once
- Using inheritance:
  - Easier to understand code
  - Make reusing and organizing code more effective
- Derivation syntax:
- Class derived-class access-specifier base-class

Inheritance - Single Inheritance



Inheritance - Multiple Inheritance

- **A class can inherit from several other classes**



Inheritance – Example

```
#include <iostream>  using namespace std;
// Base class
class Shape {
   public:
      void setWidth(int w) {  width = w;
      }
      void setHeight(int h) {  height = h;
      }
   protected:
      int width;  int height;
};
```

```
// Derived class
class Rectangle: public Shape {
public:
      int getArea() {
                  return (width *  height);
      }
};

int main(void) {
   Rectangle Rect;  Rect.setWidth(5);
   Rect.setHeight(7);
   // Print the area of the  object.
   cout << "Total area: " << Rect.getArea() << endl;
   return 0;
   }
```

Inheritance and Class Access Type

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

- Questions: What methods of base class, the derived does not inherit?
  - Constructors, destructors and copy constructors of the base class
  - Overloaded operators of the base class
  - The friend functions of the base class

- Exercise: inherit2

Inheritance and Class Access Type (cont.)

- Public Inheritance: When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class
- Protected Inheritance: When deriving from a protected base class, public and protected members of the base class become protected members of the derived class
- Private Inheritance: When deriving from a private base class, public and protected members of the base class become private members of the derived class
- Question: How about the subclasses of a subclass?

Polymorphism

- **Polymorphism** means that the same thing can exist in two forms. It has the ability to call different functions by just using one type of function call
- A **polymorphic type** is a type whose operations can also be applied to values of some other type, or types
- Example: + operator:

  4 + 5 <-- integer addition

  3.14 + 2.0 <-- floating point addition s1 + "bar" <-- string concatenation!

Different types of Polymorphism

- There are several fundamentally different kinds of polymorphism: *ad hoc, parametric, and subtype polymorphism*
  - *Ad hoc polymorphism:*
    - If a function denotes different and potentially heterogeneous implementations depending on a limited range of individually specified types and combinations, it is called *ad hoc polymorphism*
  - *Subtype/inclusion polymorphism*:
    - It is a concept wherein a name may denote instances of many different classes as long as they are related by some common superclass
    - In OOP the term 'polymorphism' is commonly used to refer solely to this *subtype polymorphism*
- Parametric polymorphism: Provides a means to execute the same code for any type
  - In C++ it is implemented via templates

Ad hoc Polymorphism

- **A kind of polymorphism in which polymorphic functions can be applied to arguments of different types**
- **Example:**

```
Program Adhoc;
 function add(x,y: Integer) : Integer;  begin          add:= x + y
                    end;
 function add(s,t: String) : String;  begin   add:= concat(s, t)
                    end;
 begin
        WriteIn (add(1,2))
        WriteIn (add('Hello', 'World!'));  end
```

- **Exercise: adhoc**

Parametric Polymorphism

- **Exercise: parametric**

Question: compile_time or runtime?

Subtype Polymorphism

- Allows a function to be written to take an object of a certain base type *B*, but also work correctly if passed an object that belongs to S, a subtype of *B*

- Exercise: poly

Question: which part presents subtype polymorphism?

Subtype Polymorphism (cont.)

Modify the code to realize subtype polymorphism

Exercise: polyv

Subtype Polymorphism (cont.)

Explain polymorphism

Exercises: feild

poly2

Virtual Functions

- A *virtual function* is a member function that is declared within a base class and redefined by a derived class
- Virtual functions implement the "one interface, multiple methods" philosophy under *polymorphism*
- Beginning a class method declaration with the keyword *virtual* in a superclass makes the function virtual for the superclass and all classes derived from the superclass

Virtual Functions (cont.)

- If a virtual method is invoked by using a reference to an object or by using a pointer to an object, the program uses the method defined for the object type rather than the method defined for the reference or pointer type. This is called *dynamic*, or *late binding*

- This behavior is important because it's always valid for a superclass pointer or reference to refer to an object of a subclass

- If you're defining a class that will be used as a superclass for inheritance, you should declare as virtual functions the class methods that may have to be redefined in subclasses

Abstract Classes & Pure Virtual Functions

- A *pure virtual function* simply acts as a placeholder that is meant to be redefined by derived classes
- It typically has a declaration and no definition (implementation)
- Classes containing pure virtual functions are termed *abstract* and they cannot be instantiated directly

```
class Shape {
  protected:
    int width, height;

  public:
    Shape( int a = 0, int b = 0) {
      width = a;
      height = b;
    }
    virtual int area() = 0;          // pure virtual function
};
```

Coercion Polymorphism

- **Coercion (implicit):**

```
double  d;
int     i;
if (d > i)      d = i;
```

- **Cast (explicit):**

```
double da = 3.3;
double db = 3.7;

int result = (int)da * (int)db  ;      //result == 9
```

Package

- A *package* is a *namespace* that organizes a set of related classes and interfaces

- Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another

- Because the software written in programming languages can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages

Package (cont.)

- **We can create package scopes using namespace declarations:**

```
namespace Engineer
{
        void Design() { … }
        // etc.
}

namespace Artist
{
        void Design() { … }
        // etc.
}
```

- **The common names in different namespaces do not conflict with each other**
- **Thus, the *Design* in *Engineer* can coexist with the *Design* in *Artist***
- **References to names outside of their namespace must be qualified using the scope resolution operator**

```
Engineer : : Design () ;
```

Package (cont.)

```
namespace Q{
  namespace V{   / /   V is a member of Q, and is fully defined within Q
/ /   namespace Q::V{   / /  alternative to the above two lines
    class C{   void m(); };  / /   C is a member of V and is fully defined within V
                          / /   C::m is only declared
    void f(); / /   f is a member of V, but is only declared  here
  }
  void V::f() / /   definition of V's member f outside of V
             / /  f's enclosing namespaces are still the global namespace, Q, and Q::V
  {
     extern void h(); / /   This declares ::Q::V::h
  }
  void V::C::m() / /  definition of V::C::m outside of the namespace (and the class body)
               / /   enclosing namespaces are the global namespace, Q, and Q::V
  {
  }
```

Package (cont.)

```
namespace Q {
  namespace V {   / /  original-namespace-definition for V  void f(); / /
    declaration ofQ::V::f
  }
  void V::f() { }   / /  OK
  void V::g() { }   / /    Error: g() is not yet a member of V  namespace V {
  / /  extension-namespace-definition for V
    void g(); / /    declaration ofQ::V::g
  }
}
namespace R {   / /    not a enclosing namespace for Q
   void Q::V::g() { }   / /    Error: cannot define Q::V::g insideR
}
void Q::V::g() { }   / /  OK: global namespace encloses Q
```