Derived Classes in C++

Professor Hugh C. Lauer
CS-2303, System Programming Concepts

(Slides include materials from *The C Programming Language*, 2nd edition, by Kernighan and Ritchie, *Absolute C++*, by Walter Savitch, *The C++ Programming Language,* Special Edition, by Bjarne Stroustrup, and from *C: How to Program*, 5th and 6th editions, by Deitel and Deitel)

Outline

- Introduction
- *Base* Classes and *Derived* Classes
- Some Examples of *Base* Class and *Derived* Class Relationships
- *Constructors* and *Destructors* in Derived Classes
- Accessing Members of *Base* and *Derived* Classes

Reading Assignment

- *Absolute C++*, Chapter 14

- A lot of similarities to *Java*
- Some differences

History

- **The whole notion of *classes*, *subclasses,* and *inheritance* came from Simula 67**
  - A generalization of notion of *records* (now known as *structs*) from *Algol*- and *Pascal*-like languages in 1960s and early 70s
  - A (very nice) programming language developed in Norway for implementing large simulation applications

Terminology

- **Inheritance is a form of *software reuse* where a new class is created to**
  - absorb an existing class's data and behaviors, and
  - enhance them with new capabilities

- **The new class, the derived class, *inherits* the members of the existing class, known as the base class**

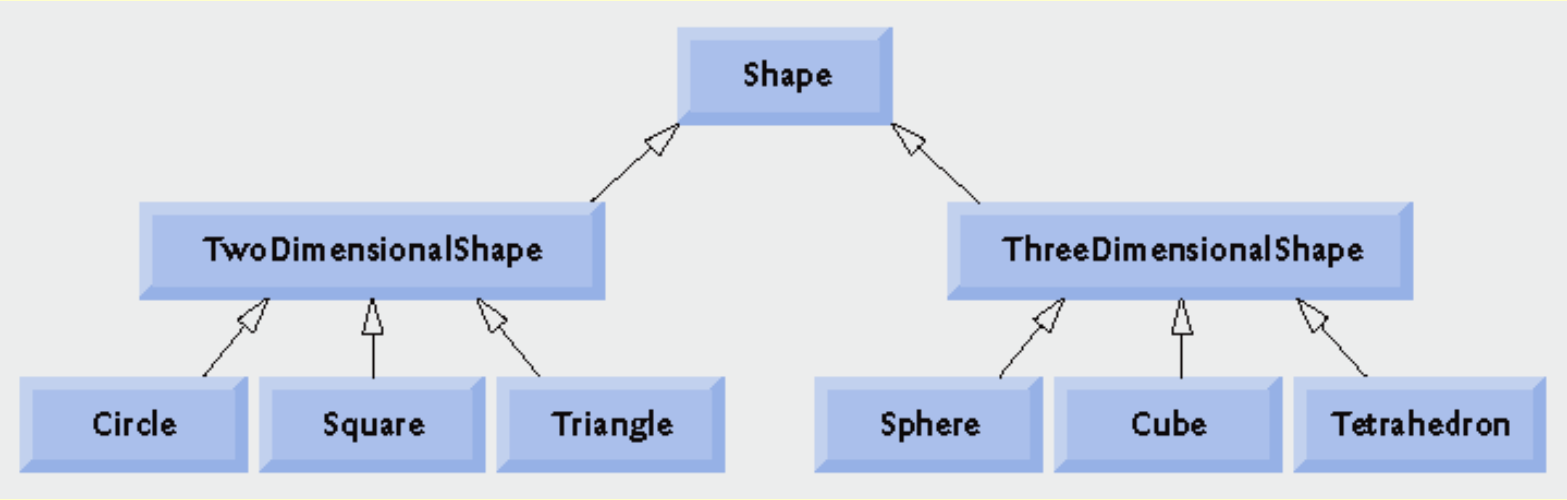
Terminology (continued)

- **C++**
  - *Derived Class*
  - *Base Class*
  - *Abstract Class*
  - *Virtual Function*
- **Java**
  - *Subclass*
  - *Superclass*
  - *Abstract Class*
  - *Abstract Method*

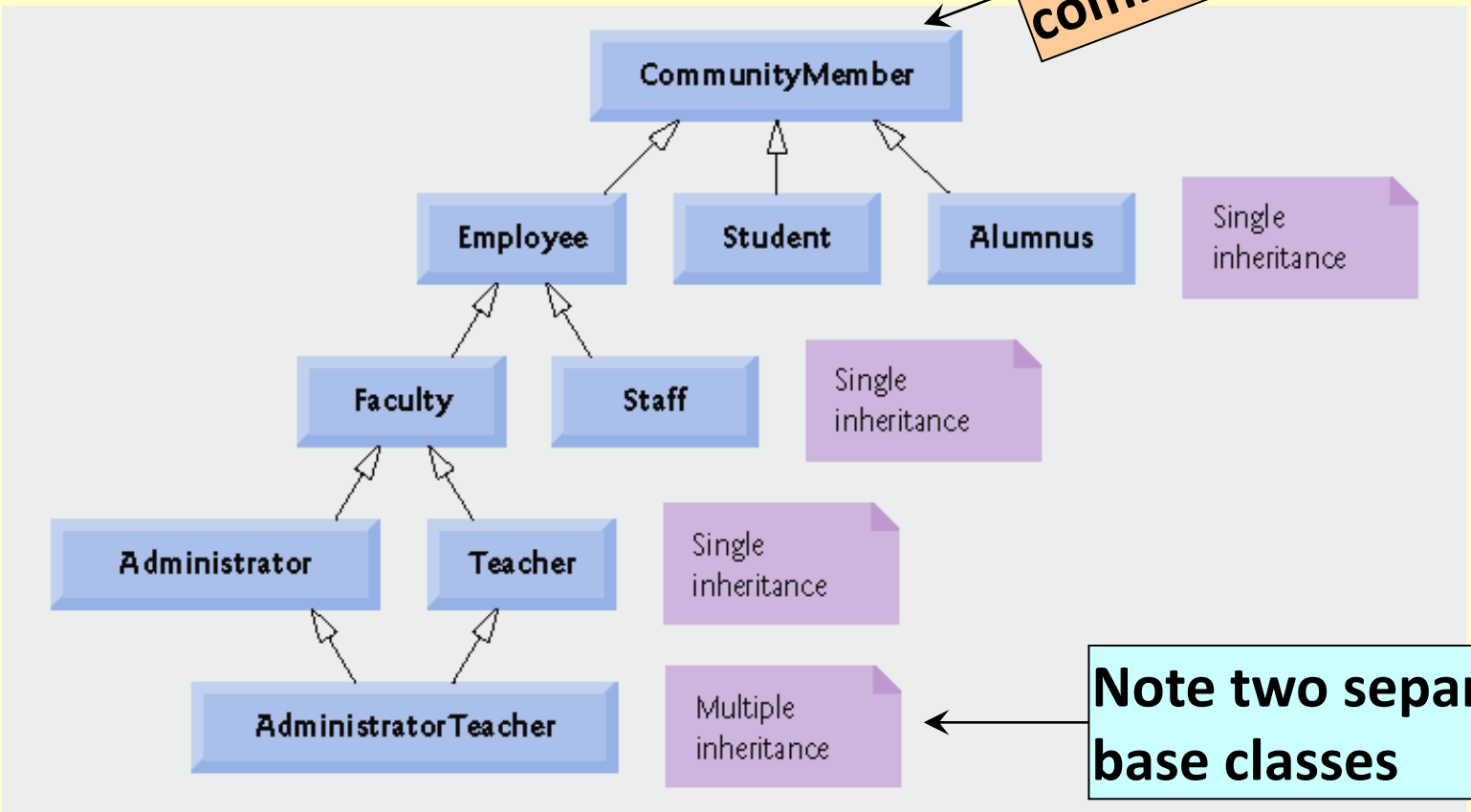
Terminology (continued)

- A *direct base class* is the base class from which a derived class explicitly inherits.
- An *indirect base class* is inherited from two or more levels up in the class hierarchy.
- In *single inheritance*, a class is derived from one base class.
- With *multiple inheritance*, a derived class inherits from multiple base classes.

Class Hierarchy



Another Class Hierarchy



Types of Inheritance in C++

- ***public:*** **every object of a derived class is also an object of its base class**
  - Note, base-class objects are NOT objects of their derived classes.
- ***private:*** **is essentially an alternative to composition**
  - I.e., derived class members not accessible from outside
- ***protected:*** **not frequently used**

Example — public class

```
class Employee {
    string givenName, familyName;
    date hiringDate;
    short department;
    ...
};

class Manager: public Employee {
    set <Employee *> group;
    short level;
    ...
}
```

Example — public class

```
class Employee {
    string givenName, familyName;
    date hiringDate;
    short department;
    ...
};

class Manager: public Employee {
    set <Employee *> group;
    short level;
    ...
}
```

Note: this is an example of the use of a *container class* from Standard Template Library

Important Note

- **Member functions of *derived class* cannot directly access *private* members of *base class***

- **Example:–**
  - *Manager* member functions in previous example *cannot* read manager's own name!
  - Because data members of a class are by default *private*

protected: Access Specifier

```
class Employee {
protected:
    string givenName, familyName;
    date hiringDate;
    short department;
    ...
};

class Manager: public Employee {
    set <Employee *> group;
    short level;
    ...
}
```

protected (continued)

- **A base class's *protected* members can be accessed by**
  - *members* and *friends* of the base class, and
  - *members* and *friends* of any class *derived* from the base class.
- **Derived-class member functions can refer to *public* and *protected* members of the base class.**
  - By simply using their names

Difference between Inheritance and Composition

- ***is-a* relationship:– *inheritance***
  - e.g., derived class object, *car*, *is* an object of the base class *vehicle*
  - e.g., derived class object, *Manager*, *is* an object of the base class *Employee*

- ***has-a* relationship:– *composition***
  - e.g., a *TreeNode* object *has* (i.e., contains) a member object of type *string*

Base Classes and Derived Classes

- **Base classes typically represent larger sets of objects than derived classes**
- **Example**
  - Base class: *vehicle*
    - Includes cars, trucks, boats, bicycles, etc.
  - Derived class: *car*
    - a smaller, more-specific subset of vehicles

Base Classes and Derived Classes (continued)

- **I.e., base classes have more objects**
  - But fewer *data* and *function members*
- **Derived classes have only *subsets* of the objects**
  - Hence the term *subclass*
  - But a derived class may have *more* members — both *data* and *function* members

Questions?

Digression – Code Re-use

- **Fundamental principle of software engineering**
- **A *body of code* is a living, evolving thing**
- **As a practical matter, *copies of code cannot keep up with each other***
- ***If you really want your hard work to support multiple purposes, applications, requirements, etc.***
  - *You really need a way for those purposes to* inherit *your code rather than copy it.*

Constructors and Destructors

- **Constructor:–**
  - Derived class constructor is called to create derived class object
  - Invokes base class constructor *first*
  - Before derived class initializer list & constructor body
  - … and so on up class hierarchy
- **Destructor:–**
  - Derived class destructor body is executed first
  - Then destructors of derived class members
  - And then destructor of base class
  - … and so on up class hierarchy

Instantiating a Derived-class Object

- **Derived-class constructor invokes base class constructor either**
  - *implicitly* (via a base-class member initializer) or
  - *explicitly* (by calling the base classes default constructor)

- **Base of inheritance hierarchy**
  - Last constructor *called* in inheritance chain is at base of hierarchy
  - Last constructor is first constructor body to *finish* executing

Example

```
class Employee {
    string name;
    int dept;
public:
    Employee(const string s, int d);
}

class Manager: public Employee {
    int level;
public:
    Manager(const string s, int d,
            int lvl);
}
```

```
Employee::Employee (const string
    s, int d):
        name(s), dept(d)
{
        …
}

Manager::Manager (const string s,
    int d, int lvl):
        Employee(s, d), level(lvl)
{
        …
}
```

Absolute C++, §14.1

Recap

- **When creating a derived-class object:–**
  - Derived-class constructor *immediately calls* base-class constructor
  - Base-class constructor executes
  - Derived class member initializers execute
  - (Finally) derived-class constructor body executes

- **If your constructor does not invoke the base class constructor explicitly, …**
- **… the compiler will generate code to invoke the base class default constructor *as first step in initialization***

Absolute C++, §14.1

Recap

- **When creating a derived-class object:–**
  - Derived-class constructor *immediately calls* base-class constructor
  - Base-class constructor executes
  - Derived class member initializers execute
  - (Finally) derived-class constructor body executes

- **This process cascades up the hierarchy if the hierarchy contains more than two levels.**

Constructors and Destructors in Derived Classes

- **Destroying derived-class objects**
  - Reverse order of constructor chain
  - Destructor of derived-class called first
  - Destructor of next base class up hierarchy is called next
  - This continues up hierarchy until the final base class is reached.
    - After final base-class destructor, the object is removed from memory

- **Base-class constructors, destructors, and overloaded assignment operators are not inherited by derived classes.**

Remember

- **When a class contains members of other classes …**
- **… *contructors* of those members are called before *class contructor* body**
  - Either in *initializer* list
  - … or by default
- **Apply same rule to members of base class!**

Remember

- Destructors for derived-class objects are called
- … in reverse order from which
- … corresponding constructors were called.

Questions?

Redefinition of Base Class Members

- **Suppose that base class has a member**
  - E.g. *void print()*
  - Knows only how to print information from base class
- **Define same member in derived class**
  - E.g. *void print()*
  - Knows how to print info for derived class
  - Needs to call base class *print()* function to print base class info

Redefinition of Base Class Members (continued)

- **Derived class**

```
void print() {
   // print derived class info

   BaseClass::print();        //prints base info

   // more derived class stuff
}
```

- **See Absolute *C++*, §14.1**

General principle:– when in derived class scope, if you need to access *anything* in base class with a naming conflict, use *'::'* scope resolution operator

Accessing Members of Base and Derived Classes

- **Let *B* be a base class with public members *m* and *n***
- **Let *D* be a derived class with public members *m* and *p***
  - I.e., *D* redefines member *m*
  - E.g., *print()* function of previous example
- **Consider the following:–**

```
B objB;
D objD;
B *ptrB;
D *ptrD;
```

Copy to white board!

Accessing Members (continued)

***objB.m* and *objB.n* are both legal**

- access members of base class

***objD.m* and *objD.p* are both legal**

- access members of derived class

***objD.n* is also legal**

- accesses member of base class!

objB.p* is *not legal

- Class *B* has no member named *p*!

Accessing Members (continued)

ptrB = new B();

***ptrB -> m and ptrB -> n* are both legal**

- access members of base class

ptrD = new D();

***ptrD -> m and ptrD -> p* are both legal**

- access members of derived class

Accessing Members (continued)

ptrB = ptrD;

- *ptrB* now points to an object of the derived class
- which by definition is an object of the base class!

***ptrB -> m* and *ptrB -> n* are both legal**

- access members of *base class* object
- *Even though* object pointed to is an object of derived class, with its own redefined member *m*!

Accessing Members (continued)

ptrB = ptrD;

***ptrB -> p* is *not* legal**

- Because *ptrB* only knows about *B*'s members!

- ***Rule:–***
  - So far, which member to access depends entirely on the type of the accessing pointer (or accessing object)
  - To bend that rule, need *polymorphism* and *virtual* members.

Questions?

Loose End:– Three Types of Inheritance in C++

- ***public:*** **every object of a derived class is also an object of its base class**
  - Note, base-class objects are NOT objects of their derived classes.
- ***private:*** **is essentially an alternative to composition**
  - I.e., derived class members not accessible from outside
- ***protected:*** **not often used**

Let D be derived from B

- **If *B* is a private base class:–**
  - I.e., *class D: private B {}*
  - Public and protected members of *B* can only be used by member and friend functions of *D*.
  - Only members and friends of *D* can convert *D** into *B**
- **Outside of member and friend functions of *D***
  - *Dptr -> p* is not allowed (where *p* is a member of *B*)
  - *Bptr = Dptr* is not allowed

Let D be derived from B (continued)

- **If *B* is a protected base:–**
  - I.e., *class D: protected B {}*
  - Public and protected members of *B* can only be used by member and friend functions of *D* and also by member and friend functions of classes derived from *D*
  - Only members and friends of *D* and **its derived classes** can convert *D** into *B**
- **I.e., outside of member and friend functions of *D* or its derived classes**
  - *Dptr -> p* is not allowed (where *p* is a member of *B*)
  - *Bptr = Dptr* is not allowed

Let D be derived from B (continued)

- **If *B* is a public base:–**
  - I.e., *class D: public B {}*
  - Public members of *B* can be used by any function
  - Protected members of *B* can be used by member and friend functions of *D* and also by member and friend functions of classes derived from *D*
  - Any function can convert *D** into *B**
- **I.e.,**
  - *Dptr -> p* is allowed (where *p* is a member of *B*)
  - *Bptr = Dptr* is allowed

Summary – Inheritance

- This topic covered the basics of *inheritance* in *C++*
- There is much, much more to say about inheritance after we cover polymorphism

Questions?

Next Topic