H.O. #11
Fall 2015
Gary Chan

Inheritance and Polymorphism

N:14; D:12,13,25

Outline

- Inheritance and Object-Oriented Design
  - Types of Inheritance
  - Building Derived Classes
  - Order of Construction and Destruction
  - Multiple Inheritance
- Polymorphism
  - Virtual Functions
  - Abstract Class

Encapsulation

- Languages such as Pascal and C facilitated development of structured programs
- Need for ability to extend and reuse software became evident
  - This leads to object-oriented programming where objects are built on top of other objects
- Data and basic operations for processing the data are encapsulated into a single "entity". This is made possible with introduction of
  - Modules
  - Libraries
  - Packages
- Implementation details are separated from class definition
  - Client code must use only public operations
  - Implementation may be changed without affecting client code

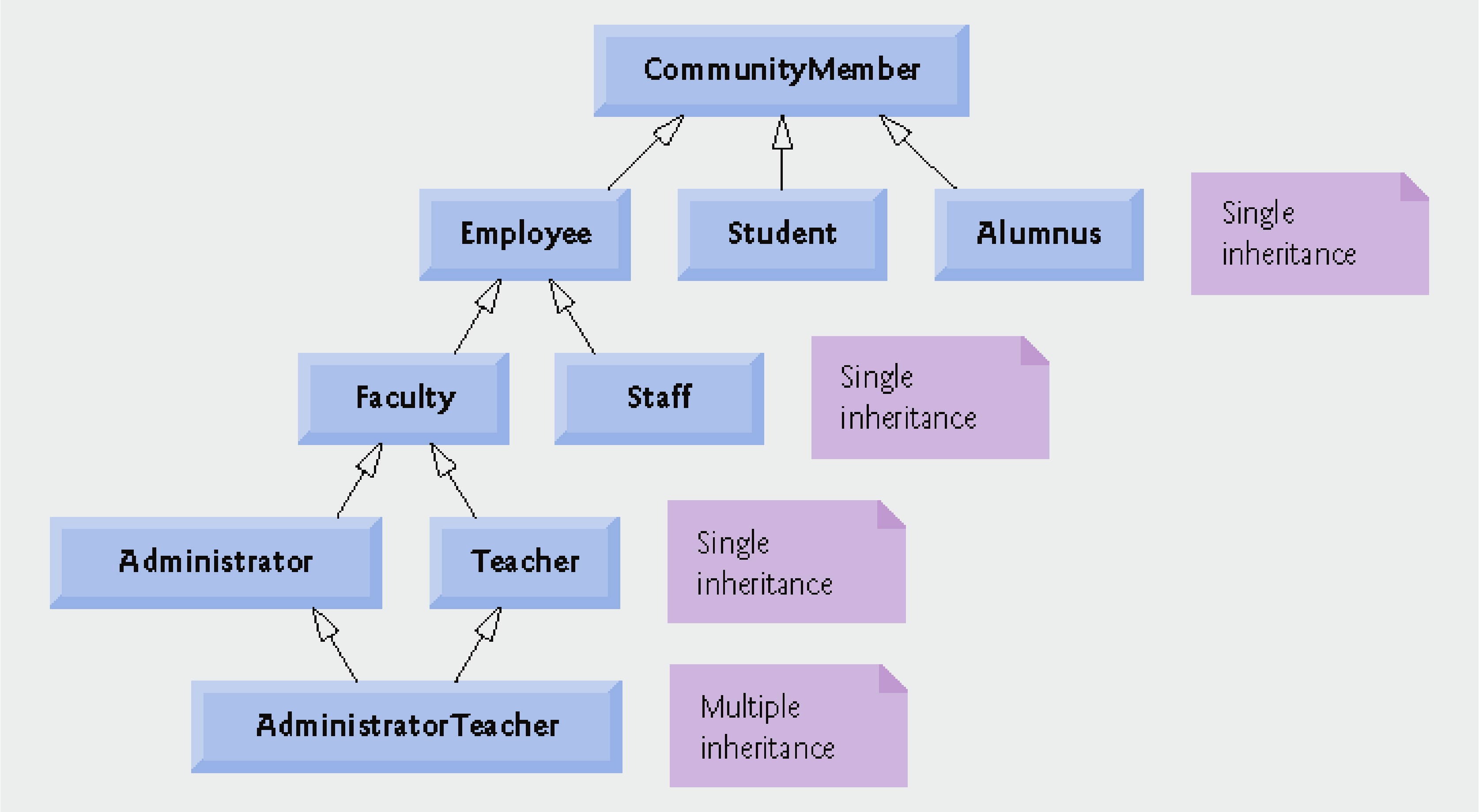
Encapsulation with Inheritance

- Some basic class features may be re-used in other classes
- A class can be *derived* from another class
  - New class inherits data and function members from the original class
  - Reusable for the new class
- Example: Consider the design of a new stack class which adds, for example `max()` and `min()`, functions to a stack
  - It is better to build "on top" of the proven stack by adding the functions
  - The new class is *inherited* or *derived* from the stack class
  - Obviously, this concept is different from creating a new class with a stack as its *member* object, because a stack cannot contain a stack

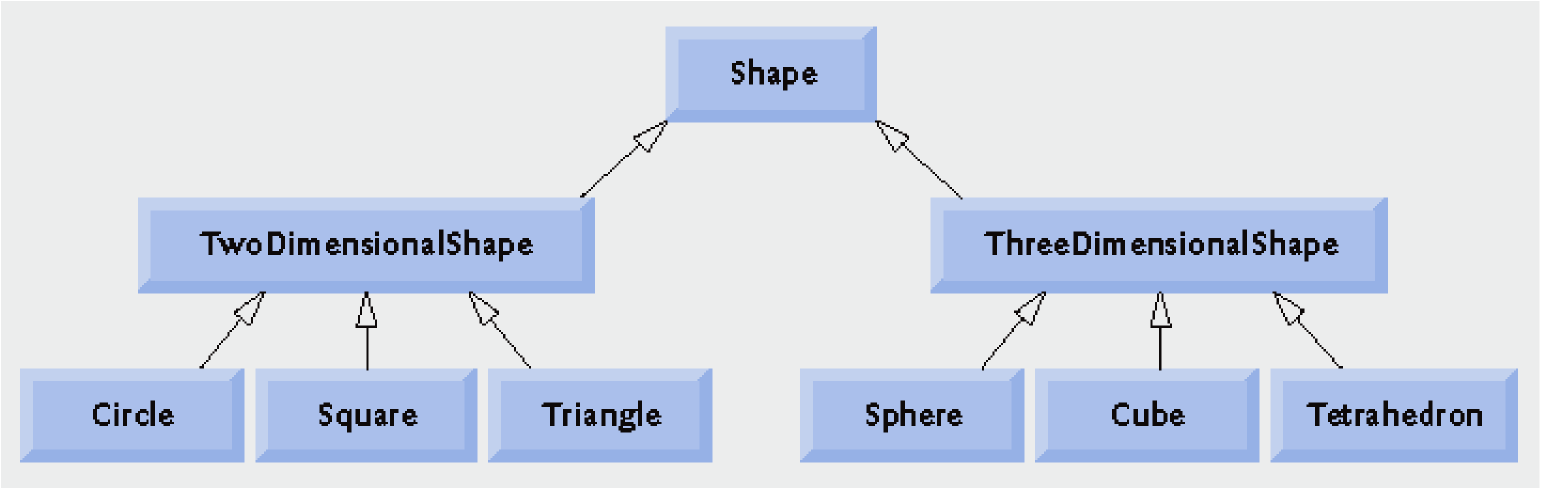
Inheritance Features and Advantages

- Software reusability
  - Often used in computer game design
- Create new class from existing class
  - Seamlessly absorb existing class's data and behaviors
  - Enhance with new capabilities
- Derived class inherits from base class
  - More *specialized* objects
  - Behaviors inherited from base class
    - Can customize
  - Additional behaviors

Inheritance Example
(Some Examples with Arrows Reversed)



Another Inheritance Example



Inheritance for Stack

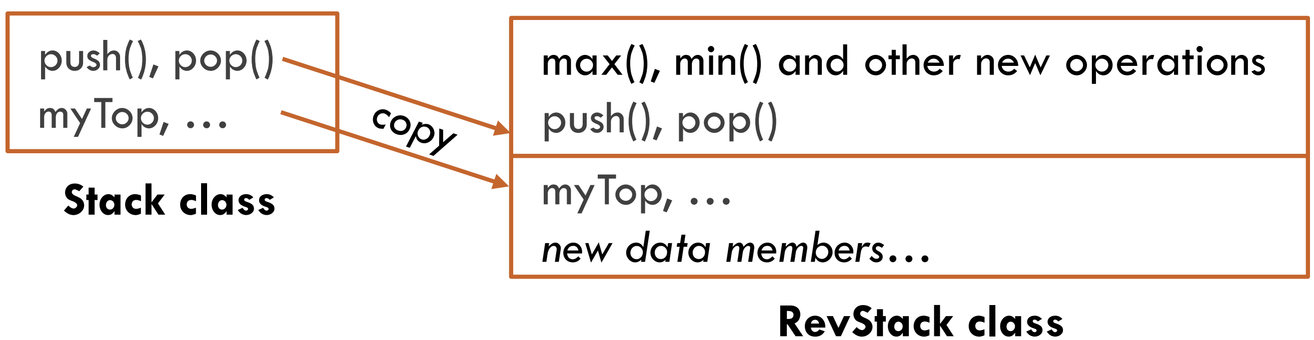
- Adapter approach
  - Build a new revised class **`RevStack`**
  - Contains **`Stack`** object as its member
- But …
  - Strictly speaking or conceptually, we cannot say anymore a **`RevStack`** is a **`Stack`**, because it contains a **`Stack`**
  - To access the functions of Stack, we need to call: RevStk.myStack.push(); for a stack we prefer simply RevStk.push()

max(), min() and other new operations including revised max() and min()
Stack myStack push(), pop()… myTop, … *new data members…*

RevStack class

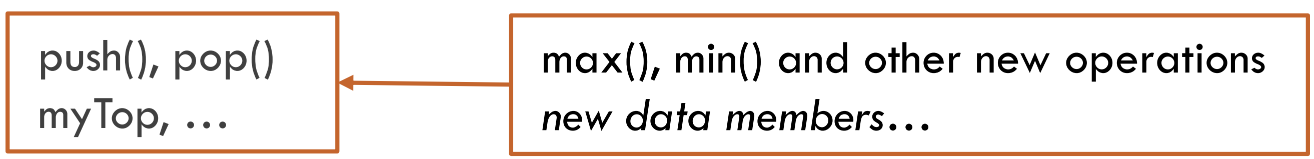
How about copy-and-paste?

- To modify the member functions, we may use copy-and-paste approach
  - Build a **`RevStack`** class (Revised Stack)
  - Copy and paste those data members and function members in **`Stack`**
  - Add the **`max()`** and **`min()`**
- Problem
  - **`RevStack`** and **`Stack`** are now separate and independent classes
  - If we update some common member functions in **`Stack`**, we must change **`RevStack`** versions



Inheritance

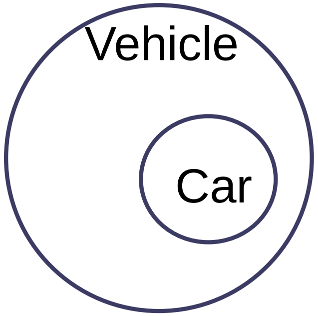
- Object-oriented approach
  - *Derive* a new class, `RevStack` from `Stack`
  - `Stack` is the base class or superclass
  - `RevStack` is a derived class or subclass
- Derived class inherits all members of base class
- Modifying `Stack` class automatically updates `Revstack` class



Stack class **RevStack class**

Relationships Between Classes

- Inheritance
  - "is-a" relationship
  - Derived class object can be treated as base class object
  - Example: Car is a vehicle
    - Vehicle properties/behaviors also apply to a car



- Composition (class in class)
  - "has-a" relationship
  - Object contains one or more objects of other classes as members
  - Example: A car object has a steering wheel object

Class-in-Class (Object-in-Object) vs. Inheritance

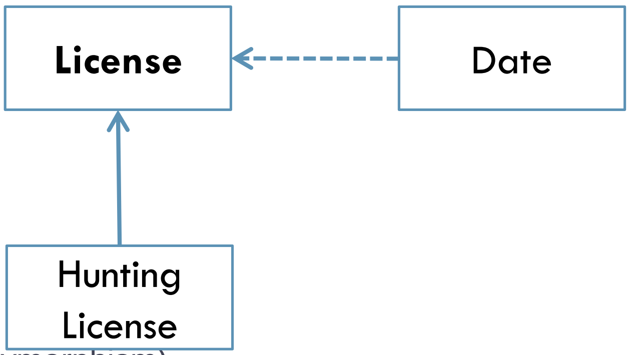
- A class declares another class as its data member, hence creating an object within another object
- Inheritance and class-in-class are two quite different things and concepts in implementation and OOP.
- Inheritance has a "is-a" relationship between derived class and base class, while class-in-class is a "has-a" relationship
- Generally, we can decide whether to use inheritance or class-in-class by common sense. If we can find some common relationship between two or more things, we should use inheritance.
  - For example, Citizen and Student with Citizen as the base class. It makes no sense to implement a Citizen class inside a Student class.
- In class-in-class, the inner class is a standalone object. Thus, the inner class and the outer class do not share the powerful features in inheritance (such as polymorphism and dynamic binding).

Relationships Between Inheritance Classes

- Base classes and derived classes
  - Object of one class “is an” object of another class
  - Example: Rectangle is a quadrilateral
    - Class Rectangle inherits from class Quadrilateral
      - Quadrilateral is the base class
      - Rectangle is the derived class
- Base class typically represents larger set of objects than derived classes
  - Base class: Vehicle
    - Includes cars, trucks, boats, bicycles, etc.
  - Derived class: Car
    - Smaller, more-specific subset of vehicles

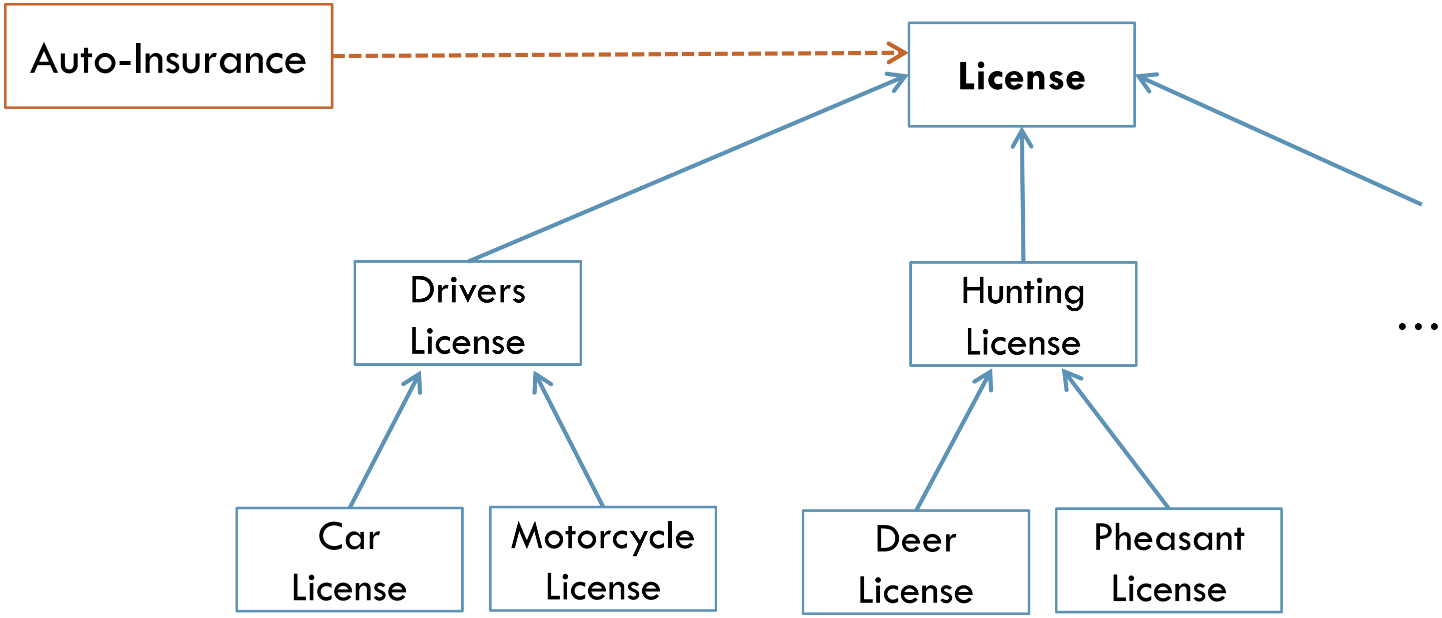
Inheritance Concept

- When class C2 is derived from class C1
  - Class C2 is-a C1
  - A **`HuntingLicense`** is-a **`License`**
  - Use public inheritance only for "is-a" relationships
- When class D1 contains a class D2 object as an element
  - D1 has-a D2
  - A **`License`** has-a **`Date`**
  - Inheritance should *not* be used for has-a relationships



"Uses-a" Relationships Between Classes

- If class D1 needs information from class D2
  - Then D1 uses-a D2
  - An **`AutoInsurance`** class needs the name from a **`DriversLicense`** class
  - May be implemented as class-in-class



Class hierarchy

- Direct base class
  - Inherited explicitly (one level up hierarchy)
  - E.g., driver licenses and license
- Indirect base class
  - Inherited two or more levels up hierarchy
  - E.g., car license and license
- Single inheritance
  - Inherits from one base class
  - E.g., the above license example
- Multiple inheritance
  - Inherits from multiple base classes
  - Base classes possibly unrelated
  - E.g., A "university student" is both a "hard-working person" and a "clever person"

Declaration of a Derived Class

```
class DerivedClassName : kind BaseClassName
{
     public:
       // new data members
     private:        // or protected
       // functions for derived class
     ...
};
```

- `kind` is one of
  - public: direct access to public region
  - private: does not allow direct access of the private region
  - protected: allowing protected region to be directly accessed by derived class and not other classes

Access Previledge of Derived Class

- Derived class inherits all members of base class
  - And members of all its ancestor classes
- Always cannot access directly *private* members of base class
- Regarding *public* and *protected* members of the upstream class, the kind of access a derived class has depends on kind of inheritance

`protected` members

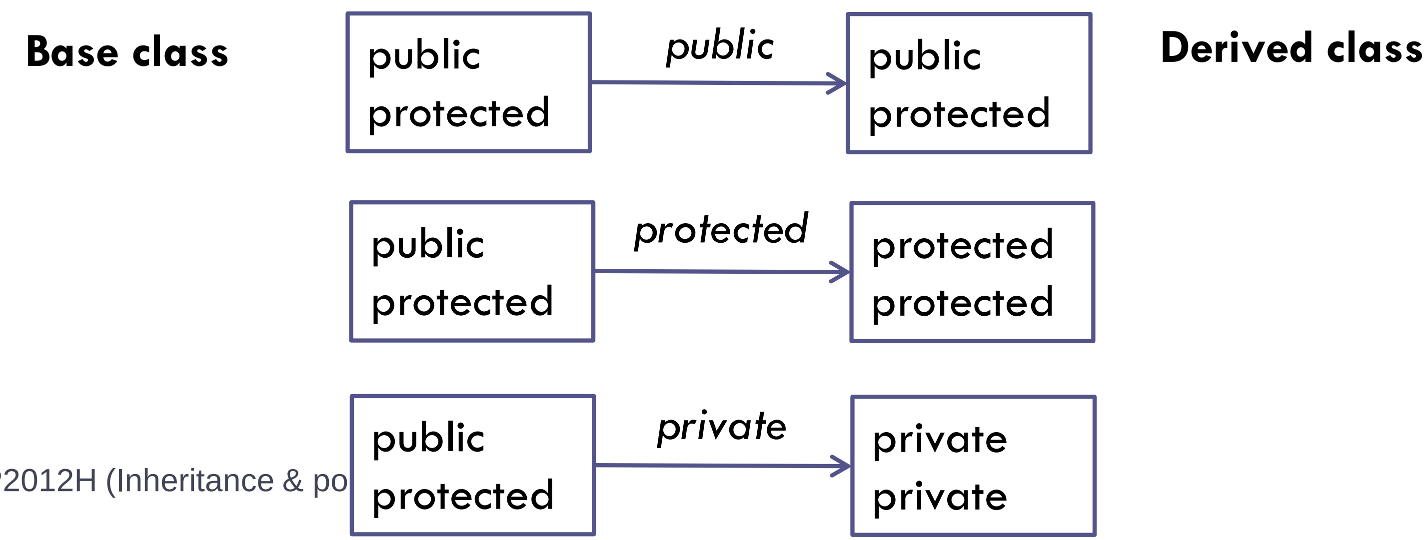
- In the base class, the `private` data members cannot be accessed by its derived class
- In the base class, the `protected` data members is like private members to other classes
  - However, the derived class can access it directly as if it is a private member
- An example of the use of `protected` keyword: `keyword_protected.cpp`

Protected Access

- Intermediate level of protection between `public` and `private`
  - For both data members and function members
  - Want the derived class to directly access members while forbid other classes to access them directly
- `protected` members in the Base class are accessible to
  - Base class members
  - Base class friends
  - Derived class members
  - Derived class friends
- Derived-class members
  - May use the public and protected members of base class
    - Simply use member names as its own members

Types of Inheritance and Region Transformation

- public inheritance (written as `class derived: public base`)
  - Base class public members → derived class public members
  - Base class protected members → derived class protected members
  - All classes can directly access the public members
  - Only the *derived* classes can *directly* access the *protected* members
- protected inheritance (written as `class derived: protected base`)
  - Base class public and protected members → derived class protected members
  - Classes in the inheritance hierarchy can still access the members (because they are protected members), but not for other classes
- private inheritance (written as `class derived: private base`)
  - Base class public and protected members → derived class private members
  - Classes in the downstream inheritance hierarchy can no longer access the members (and neither can all the other classes)



Types of Inheritance and Member Access

Base-class member-access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
public	public in derived class. Can be accessed directly by member functions, friend functions and nonmember functions.	protected in derived class. Can be accessed directly by member functions and friend functions.	private in derived class. Can be accessed directly by member functions and friend functions.
protected	protected in derived class. Can be accessed directly by member functions and friend functions.	protected in derived class. Can be accessed directly by member functions and friend functions.	private in derived class. Can be accessed directly by member functions and friend functions.
private	Hidden in derived class. Can be accessed by member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by member functions and friend functions through public or protected member functions of the base class.

```
class derived: public base
```

```
class derived: protected base
```

```
class derived: private base
```

`kind.cpp`: Illustration of kinds of inheritance

- Note how the protected members are accessed in different derived classes
- Note how the public and protected members of Base class are changed by the kind of inheritance. Their accessibility is also changed.

Public Inheritance

- Specify with:

```
class TwoDimensionalShape : public Shape
```

  - Class TwoDimensionalShape inherits from class Shape
- Base class private members
  - Not accessible directly (still inherited)
  - Manipulated through inherited public member functions
- Base class public and protected members
  - Inherited with original member access
- friend functions
  - *Not* inherited

Building Derived Classes (for `public` Inheritance)

- Derived class constructors
  - Use parent class's constructors to initialize base class members
  - Is actually a call to the base class constructor
  - The member-initializer list initializes member objects
  - Need to explicitly invoke base-class constructors in the member initializer
- Accessing inherited data members
  - If base class data public, derived class can access, even alter it
  - If base class protected, can also alter it directly
  - If base class data private, must use accessor functions

Building Derived Classes: Reusing Operations

- Derived class may extend or replace base class function of the same name
- Possible to call the base class function with scope resolution operator

```
void DerivedClass::foo()
{   // extending the base class function
        . . .
        BaseClass::foo();
        . . .
}
```

CommissionEmployee Example

- CommissionEmployee
  - First name, last name, SSN (Social Security Number, i.e., ID), commission rate, gross sale amount
- BasePlusCommissionEmployee
  - CommissionEmployee: First name, last name, SSN, commission rate, gross sale amount
  - And also base salary
- `Class` BasePlusCommissionEmployee
  - Much of the code is similar to CommissionEmployee
  - Additions
    - private data member baseSalary
    - Methods setBaseSalary and getBaseSalary

CommissionEmployee Example: Class BasePlusCommissionEmployee

- Derived from class CommissionEmployee
  - Is a CommissionEmployee
  - Inherits all public members
  - Use base-class initializer syntax to initialize base-class data member
- Has data member baseSalary
- Base class implementation
  - `CommissionEmployee1.h`, `CommissionEmployee1.cpp`
- Derived class implementation
  - `BasePlueCommissionEmployee1.h`, `BasePlusCommissionEmployee1.cpp`
- **Compilation *error* because derived class cannot directly access private members of `CommissionEmployee` class in `print()` and `earnings()`**

Protected Access

- Use `protected` keyword to fix the problem
- `CommissionEmployee2.h, CommissionEmployee2.cpp, BasePlusCommissionEmployee2.h, BasePlusCommissionEmployee2.cpp, test2.cpp`

tester2.cpp Sample Output

```
Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

Using Protected Data Members

- Advantages
  - Derived class can modify values directly
  - Avoid set/get method call overhead → Slight increase in performance
- Disadvantages
  - No validity checking: Derived class can assign illegal value to protected members
  - Implementation becomes dependent on the base class
    - Derived class functions becomes very dependent on base class implementation
    - Using protected access, base class implementation changes may result in derived class modifications, e.g., a change of the name in the protected region of the base class may leads to many changes in the derived class
    - This leads to fragile (brittle) software

Best Software Engineering Practice

- Declare data members as private
- Provide public get and set functions
- Use get and set method to obtain and set values of data members
- `CommissionEmployee3.h`, `CommissionEmployee3.cpp`, `BasePlusCommissionEmployee3.h`, `BasePlusCommissionEmployee3.cpp`, `tester3.cpp`

tester3.cpp Sample Output

```
Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

Remarks

- Using a member function to access a data member's value can be slightly slower than accessing the data directly.
  - But programmers should write code that adheres to proper software engineering principles, and leave optimization issues to the compiler
- In function over-riding, failure to use the scope **::** operator prefixed with the name of the base class when referencing the base class's member function causes *infinite* recursion (as the derived-class member function calls itself)
  - E.g., The `earnings()` function in the base class is overridden by

```
double BasePlusCommissionEmployee::earnings() const
```

in BasePlusCommissionEmployee3.cpp

Base and Derived Functions: Function Over-riding, not Over-loading/co-existing functions

- In the derived class, having a member function with the same name as a base class function hides (or overrides) the base-class version of the function
- It is OK to call `d.print()` if `print` were *not* defined at all in the `Derived` class (prints out `base`)

```
class Base{
public:
  void print(){cout << "base\n";}
};

class Derived: public Base{
public:
  void print( int i ){ cout << i << " Derived\n";}
  void print( char ch ){ cout << ch << " Derived\n";}
};

int main(){
  Derived d;
  d.print( 2 );  // print 2 Derived
  d.print('d');  // print d Derived
  // d.print();    Not O.K.: no matching function for Derived::print()
  return 0;
}
```

Order of Construction

- Called at the initializer list, e.g., `Derived:: Derived(): Base(){…}`
  - The base class MUST be constructed in the initializer
  - You MUST only call the immediate/direct base class constructor
- Chain of constructor calls to instantiate derived-class object:
  - Derived-class constructor invokes base class constructor
    - Implicitly or explicitly
  - Base-class constructor: Base of inheritance hierarchy
    - Like a recursive stack
  - Initializing data members
    - Each base-class constructor initializes its data members that are inherited by derived class
- When a program creates a derived-class object:
  1. The derived-class constructor immediately calls the base-class constructor
  2. The base-class constructor's *body* (i.e., within { }) executes
  3. Then the derived class's member initializer list execute
  4. Finally the derived-class constructor's body executes
- This process cascades up the hierarchy if the hierarchy contains more than two levels in a recursive manner

Order of Destruction

- Chain of destructor calls to destroy derived-class object:
  - Reverse order of constructor chain
  - Destructor of derived-class is called first
  - Destructor of the next base class up hierarchy next
    - Continue up hierarchy until final base reached
      - After final base-class destructor, object is removed from memory
- Base-class constructors, destructors, assignment operators
  - Not inherited by derived classes
- Example on order of construction and destruction
  - `CommissionEmployee4.h, CommissionEmployee4.cpp, BasePlusCommissionEmployee4.h, BasePlusCommissionEmployee4.cpp, order.cpp`

order.cpp Sample Output (1/4)

```
CommissionEmployee constructor:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

CommissionEmployee destructor:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04


CommissionEmployee constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
```

CommissionEmployee constructor called for object in block; destructor called immediately as execution leaves scope

Base-class CommissionEmployee constructor executes first when instantiating derived-class BasePlusCommissionEmployee object

order.cpp Sample Output (2/4)

```
BasePlusCommissionEmployee constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00
```

Derived-class BasePlusCommissionEmployee constructor body executes after base-class CommissionEmployee's constructor finishes execution

```
CommissionEmployee constructor:
commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
```

Base-class CommissionEmployee constructor executes first when instantiating derived-class BasePlusCommissionEmployee object

order.cpp Sample Output (3/4)

```
BasePlusCommissionEmployee constructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00

BasePlusCommissionEmployee destructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00

CommissionEmployee destructor:
commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
```

Derived-class BasePlusCommissionEmployee constructor body executes after base-class CommissionEmployee's constructor finishes execution

Destructors for BasePlusCommissionEmployee object called in reverse order of constructors

order.cpp Sample Output (4/4)

```
BasePlusCommissionEmployee destructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00

CommissionEmployee destructor:
commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
```

Destructors for BasePlusCommissionEmployee object called in reverse order of constructors

Multiple Inheritence

- When a derived class inherits members from two or more base classes
  - Provide comma-separated list of base classes after the colon following the derived class name
- Can cause ambiguity problems
  - Should be used only by experienced programmers
  - Newer languages do not allow multiple inheritance
  - A common issue occurs if more than one base class contains a member with the same name
    - Solved by using the binary scope resolution operator

Multiple Inheritence (Cont.)

- Should be used when an "is a" relationship exists between a new type and two or more existing types
  - i.e. type A "is a" type B and type A "is a" type C
- Can introduce complexity into a system
  - Great care is required in the design of a system to use multiple inheritance properly
  - Should not be used when single inheritance and/or composition will do the job
- Example:
  - `Base1.h, Base2.h, Derived.h, Derived.cpp, multiple.cpp`

multiple.cpp Sample Output

- Note the use of base-class pointer pointing to a derived-class objects
  - Invoking the member function of the derived object

```
Object base1 contains integer 10
Object base2 contains character Z
Object derived contains:
    Integer: 7
  Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
    Integer: 7
  Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Size of the Base-class and Derived-class Objects

- The size of a derived object is not the sum of the base-class object and derived-class members
  - Probably due to memory alignment and internal representation of derived-class object
- The size of the derived-class object that a base-class handle points to is actually that of the base-class object.

```
#include <iostream>
using namespace std;

class base{
public:
  int i;    // 4 byes
  float f;  // 4 bytes
};

class derived: public base{
public:
  double d;      // 8 bytes
  double *dptr;  // 8 bytes
  char c[100];   // 100 bytes
};
```

```
int main(){

  // for base object
  cout << sizeof (int) << endl;
  cout << sizeof (float) << endl;
  cout << sizeof (base) << endl << endl;

  // for derived object
  cout << sizeof (double) << endl;
  cout << sizeof (double *) << endl;
  cout << sizeof (char [100]) << endl;
  cout << sizeof (derived) << endl << endl;

  base *bptr = new derived;
  cout << sizeof (*bptr) << endl;

  derived *dptr = new derived;
  cout << sizeof (*dptr) << endl;

  return 1;
}
```

```
4
4
8

8
8
100
128

8
128
```

Software Engineering: Customizing Existing Software with Inheritance

- Inheriting from existing classes
  - Can include additional members
  - Can redefine base-class members
  - No need to have direct access to base class's source code
    - Only need to link to object code
- Good for those independent software vendors (ISVs)
  - Develop proprietary code for sale/license
    - Available in object-code format
  - Users derive new classes
    - Without accessing ISV proprietary source code

Polymorphism

Polymorphism and Dynamic Binding

- "Polymorphic" behavior in functions and classes
  - Function name can be overloaded
  - Function template is a pattern for multiple functions
  - Class template is a pattern for multiple classes
- In these cases the compiler determines which version of the function or class to use *during* the compilation time
  - Called static or early binding
- Sometimes we don't know the kind of object until run time
  - Dynamic binding
  - Usually involves pointers to some objects which are not known beforehand

Polymorphism with inheritance hierarchies

- "Program in the general" vs. "program in the specific"
- Process objects of classes that are part of the same hierarchy as if they are objects of a single class
  - E.g., vehicles ← 4-wheel vehicle ← passenger car ← sport car
  - Objects can be created in any part of the chain of hierarchy
- Each object performs the correct tasks for that object's type
  - Different actions occur depending on the type of object
- New classes can be added with little or no modification to existing code

Using Handles

- A handle is a variable whose value is the *address* of that object
  - It is a pointer variable (address of the object)
  - Refers to the object indirectly
- Handle for *base* class object can also refer to any *derived* class object (`SalariedEmployee` is derived from `Employee`)

  ```
  Employee * eptr;  // handle
  eptr = new Employee();
  ```
  or
  ```
  eptr = new SaleriedEmployee(); // o.k.!
  ```
- Then `eptr->display(cout);` will always work
  - It always calls *Employee's* member function `display` if it is implemented as an actual function, even if it is pointing to SalariedEmployee object

Invoking Functions

- Cannot aim derived-class pointer to a base-class object
- Aim base-class pointer at base-class object
  - Invoke base-class functionality
- Aim derived-class pointer at derived-class object
  - Invoke derived-class functionality
- Aim base-class pointer at *derived-class* object
  - Can *only* invoke base-class functionalities
  - Because derived-class object is an (inherited) object of base class
- Invoked functionality depends on the handle type used to invoke the function (which is base or derived object).
  - Therefore, if the handle is base pointer, even if it points to a derived-class object, it invokes the functionality of *base* class
- `CommissionEmployee1.h`, `CommissionEmployee1.cpp`, `BasePlusCommissionEmployee1.h`, `BasePlusCommissionEmployee1.cpp`, `tester1a.cpp`

tester1a.cpp Sample Output (1/2)

```
Print base-class and derived-class objects:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Calling print with base-class pointer to
base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
```

tester1a.cpp Sample Output (2/2)

```
Calling print with derived-class pointer to
derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

Invoking Functions

- The pointer must be a *base*-class pointer, pointing to a derived-class object
  - All the base class functions of the derived object can be called. This is not a problem because derived class inherits all the functions from the base class.
  - Because it is a base class pointer, cannot access the members of derived-class even if the base-class pointer is pointing to the derived-class object
- Aim a *derived*-class pointer at a *base*-class object is an error
  - C++ compiler generates error
  - This is because
    - A derived-class pointer is supposed to be able to access all the derived-class member functions that it points to
    - If the pointer is pointing to a base class, some of these derived-class functions may not even be available at the base class

Summary of the Allowed Assignments

- Four ways to aim base-class and derived-class pointers at base-class and derived-class objects

	Base object	Derived object
Base pointer	Straightforward	Is safe, but can be used to invoke only member functions that *base*-class declares; Can achieve polymorphism with `virtual` function
Derived pointer	Compilation error	Straightforward

Polymorphism and Dynamic Binding

- So far, we have seen how a base-class handle can bind dynamically to a derived-class object
  - But the functions that can be used are still of the base-class
- We want to call the functions of the *derived* class
- Example: Animal hierarchy
  - `Animal` base class – every derived class has function `move`
  - Different animal objects maintained as a vector of `Animal` pointers
  - Program issues same message (`move`) to each animal generically
  - Proper function gets called
    - A `Fish` will `move` by swimming
    - A `Frog` will `move` by jumping
    - A `Bird` will `move` by flying
- Another example: Computer games
  - Different characters, if hit, may have their scores updated differently (using, e.g., an `update_score()` function)

Virtual Functions and Dynamic Binding

- Which version is called must be deferred to run time
  - This is dynamic or *late binding*
- Accomplished with *virtual functions*
  - Each object contains some virtual function
  - Compiler creates a virtual function table (vtbl) for each object
  - Table of pointers to actual codes of the required function (e.g., move), which is to the actual function implementation of the derived class
  - Make it possible to invoke the object type's functionality (the actual derived class object), rather than invoke the handle type's (i.e., the type of the pointer) functionality
  - Crucial to implementing polymorphic behavior

Virtual Functions

- Normally handle determines which class's functionality to invoke
  - If it is of base-class pointer, base member functions will be invoked even though the object that it points to is a derived class
- With virtual functions
  - Type of the *object* being pointed to, not type of the *handle*, determines which version of a virtual function to invoke
  - Allows program to dynamically (at runtime rather than compile time) determine which function to use
  - Dynamic binding or late binding
- Declared by preceding the function's prototype with the keyword virtual in base class
- Derived classes override function as appropriate
  - Replacing the function
  - A call to the function will use the definition of the derived class

Virtual Functions (Cont.)

- Once declared `virtual`, a function remains virtual all the way down the hierarchy
  - Even so, as a good software practice, you should put `virtual` to all the functions you want to make virtual
- Static binding
  - When calling a virtual function using specific object with dot operator, function invocation is resolved at compile time
  - E.g., `obj.virtual_function(); // known obj type at compilation`
- Dynamic binding
  - Dynamic binding occurs only for pointer and reference handles when the objects that these handles point to are not known at compile time
- CommissionEmployee2.h, CommissionEmployee2.cpp, BasePlusCommissionEmployee2.h, BasePlusCommissionEmployee2.cpp, test2er.cpp
  - Note the use of `virtual` keyword in both base and derived classes

tester2.cpp Sample Output (1/3)

```
Invoking print function on base-class and derived-class
objects with static binding

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Invoking print function on base-class and derived-class
objects with dynamic binding
```

tester2.cpp Sample Output (2/3)

```
Calling virtual function print with base-class pointer
to base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Calling virtual function print with derived-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

tester2.cpp Sample Output (3/3)

```
Calling virtual function print with base-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Determining the Type of Object Using `dyanmic_cast`

- `dynamic_cast` **can be used only with pointers and references to base class objects. Its purpose is to ensure that the result of the type conversion is a valid *complete* object of the requested class.**
  - Return NULL is not so

```
#include <iostream>
#include <typeinfo>
#include <string>

using namespace std;

class base{
public:
  virtual void print(){ cout << "Base object\n";}
};

class derived: public base{
public:
  virtual void print(){ cout << "Derived object\n"; }
};

int main(){

  base * bptr[ 2 ];
  // check whether it points to a derived obj
  derived * is_derived;
  bptr[ 0 ] = new base();
  bptr[ 1 ] = new derived();
```

```
 // check whether the pointer can be successfully cast
  is_derived = dynamic_cast< derived * > (bptr[ 0 ]);

  if( is_derived )
    cout << "bptr[0] is a derived object.\n";
  else
    cout << "bptr[0] is a base object.\n";

  is_derived = dynamic_cast< derived * > (bptr[ 1 ]);

  if( is_derived )
    // derived class
    is_derived -> print();    // call derived functions
  else
    // is_derived is NULL; base class
    bptr[ 1 ] -> print();     // call base functions

  return 0;
}
```

bptr[0] is a base object.
Derived object

Abstract and Concrete Classes

- Classes from which the programmer never intends to instantiate any objects
  - Incomplete—derived classes must define the “missing pieces” or “missing parts”
  - Too generic to define any real objects out of it
- Normally used as base classes, called abstract base classes
  - Provides an appropriate base class from which other classes can inherit
  - Classes used to instantiate objects are called concrete classes
    - Must provide implementation for every member function they define

Pure Virtual Functions

- A class is made abstract by declaring one or more of its virtual functions to be "pure"
  - No object can be created out of it
  - Placing "= 0" in its declaration
  - Example: `virtual void draw() const = 0;`
    - "= 0" is known as a pure specifier
- Do not provide implementations
  - Every concrete derived class *must* override all base-class pure virtual functions with concrete implementations
  - If not overridden, derived-class will also be abstract
- Used when it does not make sense for base class to have an implementation of a function, but the programmer wants all concrete derived classes to implement the function

Abstract Classes and Pure Virtual Functions

- We can use the abstract base class to declare pointers and references
  - Can point to objects of any concrete class derived from the abstract class
  - Programs typically use such pointers and references to manipulate derived-class objects polymorphically
- Polymorphism is particularly effective for implementing software systems
  - E.g., reading or writing data from and to different devices of the same base class
- Iterator class (using base class pointer)
  - Can traverse all the objects in a container

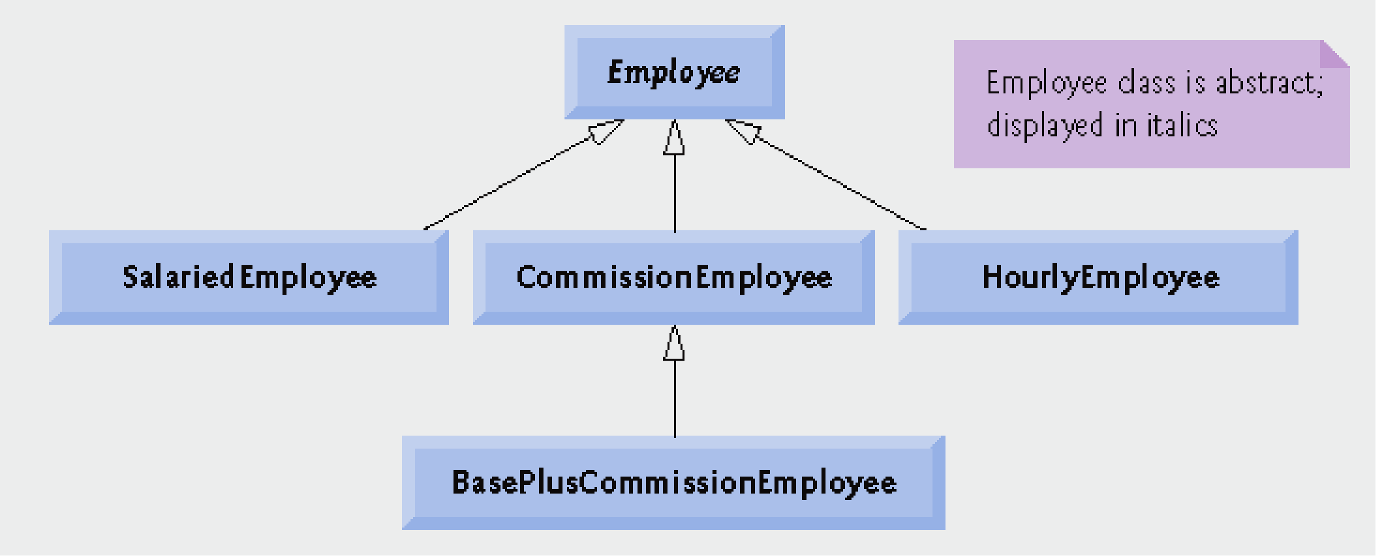
```cpp
#include <iostream>

using namespace std;

class base{
public:
  virtual void print() = 0;
  virtual void print2() = 0;
};

class derived1: public base{
public:
  virtual void print(){
    cout << "derived1\n";
  }
  virtual void print2(){}  // must have this line,
          // otherwise compiler complains in main()
};

class derived2: public base{
public:
  virtual void print(){
    cout << "in derived2\n";
  }
  // do not need to define print2() here as
  // derived2 is not a concrete class
};

class derived3: protected derived2{
public:
  virtual void print2(){
    cout << "In derived3\n";
  }
};
```

```cpp
int main(){
  derived1 d1;
  // derived2 d2; compiler complains:
  // the following virtual functions are abstract:
  // void base::print2()
  derived3 d3;

  d1.print();
  // d3.print();  print() is inaccessible; ok if
public inheritance
  d3.print2();

  base * bptr1 = new derived1();  // ok
  //  base * bptr2 = new derived3();
  // base is an inaccessible base of derived3

  //  derived2 *d2ptr = new derived3();
  // derived2 is an inaccessible base of derived3


  return 1;
}
```

derived1
In derived3

Case Study: Payroll System Using Polymorphism

- Enhanced CommissionEmployee-BasePlusCommissionEmployee hierarchy using an abstract base class
- Abstract class Employee represents the general concept of an employee
  - Declares the "interface" to the hierarchy
  - Each employee has a first name, last name and social security number
- Earnings calculated differently and objects printed differently for each derived class



Creating Abstract Base Class Employee

- Provides various get and set functions
- Provides functions `earnings()` and `print()`
  - Function `earnings()` depends on type of employee, so declared pure virtual
    - Not enough information in class Employee for a default implementation
  - Function `print()` is virtual, but not pure virtual
    - Default implementation provided in Employee
- Example maintains a vector of Employee pointers
  - Polymorphically invokes proper earnings and print functions

Polymorphic Interface

	earnings	print
Employee	= 0	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* (40 * wage) + ((hours - 40) * wage * 1.5)	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	baseSalary + (commissionRate * grossSales)	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Creating Concrete Derived Class

- SalariedEmployee inherits from Employee
  - Includes a weekly salary
    - Overridden earnings function incorporates weekly salary
    - Overridden print function incorporates weekly salary
  - Is a concrete class (implements all pure virtual functions in abstract base class)

SalariedEmployee.h

```
class SalariedEmployee : public Employee {
public:
   SalariedEmployee( const string &, const string &,
      const string &, double = 0.0 );
   void setWeeklySalary( double ); // set weekly salary
   double getWeeklySalary() const; // return weekly salary

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print SalariedEmployee object
private:
   double weeklySalary; // salary per week
};
```

- SalariedEmployee inherits from Employee, must override `earnings` to be concrete
- Functions `earnings` and `print` in the base class will be overridden (`earnings` defined for the first time)

Creating Indirect Concrete Derived Class

- BasePlusCommissionEmployee inherits from CommissionEmployee
  - Includes base salary
    - Overridden `earnings()` function that incorporates base salary
    - Overridden `print()` function that incorporates base salary
  - Concrete class
    - Not necessary to override `earnings()` to make it concrete, can inherit implementation from `CommissionEmployee`
    - Although we do override `earnings()` to incorporate base salary

Demonstrating Polymorphic Processing

- Create objects of types SalariedEmployee, HourlyEmployee, CommissionEmployee and BasePlusCommissionEmployee
  - Demonstrate manipulating objects with static binding
    - Using name handles rather than pointers or references
    - Compiler can identify each object's type to determine which print and earnings functions to call
  - Demonstrate manipulating objects polymorphically
    - Uses a vector of *Employee* pointers
    - Invoke virtual functions using pointers and references
- One may also "cast" a derived object to its base class:

```
Base b = derived_obj;
```

payroll.cpp Sample Output (1/3)

```
Employees processed individually using static binding:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

payroll.cpp Sample Output (2/3)

```
Employees processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

payroll.cpp Sample Output (3/3)

```
Virtual function calls made off base-class references:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00


commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Last Test: What is the Output? (1)

```
#include <iostream>
using namespace std;

class A {
public:
  A() {}
  void f() {cout << "A::f()" << endl;}
};

class B: public A {
public:
  B() {}
  void f() {cout << "B::f()" << endl;}
};

class C: public B {
public:
  C() {}
  void f() {cout << "C::f()" << endl;}
};
```

```
int main(){
  A* z = new A;
  z->f();
  delete z;

  A* x = new B;
  x->f();
  delete x;

  A* y = new C;
  y->f();
  delete y;
  return 0;
}
```

Output:
A::f()
A::f()
A::f()

Last Test: What if we add virtual to class A (and everything else remains the same)?

```
class A {
public:
  A() {}
  virtual void f() {cout << "A::f()" << endl;}
};
```

Output:
A::f()
B::f()
C::f()

Virtual Destructors

- Nonvirtual destructors
  - Destructors that are not declared with keyword virtual
  - If a derived-class object is destroyed explicitly by applying the `delete` operator to a *base-class pointer* to the object, the behavior is undefined
  - This is because `delete` may be applied on a base-class object, instead of the derived class
- virtual destructors
  - Declared with keyword `virtual`
    - That means that all derived-class destructors are virtual
  - With that, if a derived-class object is destroyed explicitly by applying the `delete` operator to a *base-class pointer* to the object, the appropriate derived-class destructor is then called
  - Appropriate base-class destructor(s) will execute *afterwards*

```
#include <iostream>
using namespace std;

class Base{
public:
  virtual ~Base() { cout <<"Base Destroyed\n"; }
};

class Derived: public Base{
public:
  virtual ~Derived() { cout << "Derived Destroyed\n"; }
};

int main(){
  Derived d;
  Base *bptr = new Derived();
  delete bptr;             // explicit delete → call the destructor immediately
  bptr = new Derived();    // the object will be deleted by garbage collection
                           // after program exits, and hence no destructor statement
  return 0;
}
```

Derived Destroyed *(for "delete bptr")*
Base Destroyed
Derived Destroyed *(for object d going out of scope)*
Base Destroyed