Inheritance

Computer Programming for Engineers
(DASF003-41)

Instructor:
Sungjae Hwang (jason.sungjae.hwang@gmail.com)

Today

- **Inheritance Basics**
  - Concept/types in OOP, base classes and derived classes,
  - Example with Employee Class
- **Constructors/destructors**
- **`protected` qualifier/inheritance**
- **Redefining member functions**
- **Multiple Inheritance**

INHERITANCE BASICS

Introduction

Object-oriented programming

- is a Powerful programming technique.
- provides abstraction dimension called ***inheritance***.

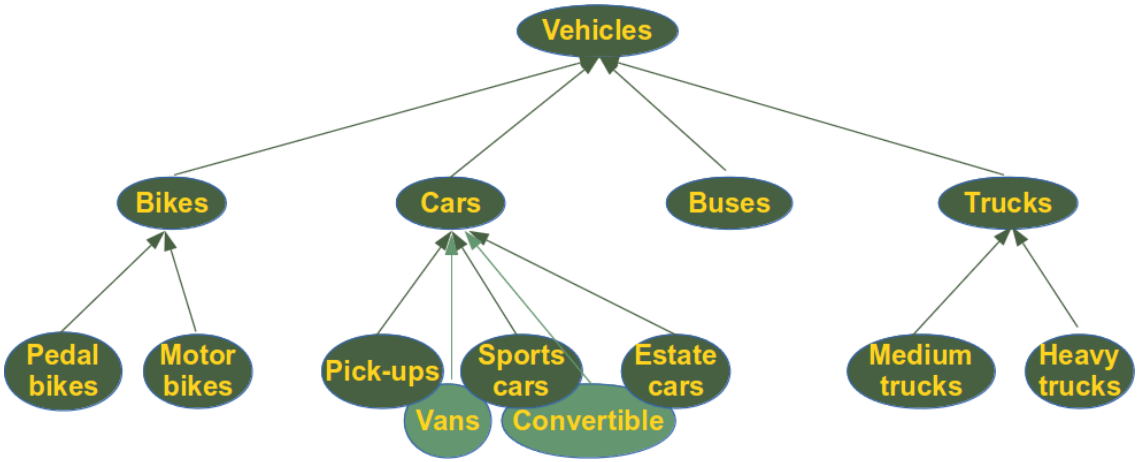
How does it work?

1. General form of class is defined
2. Specialized versions then inherit properties of general class
3. And add to it/modify it's functionality for it's appropriate use

Example

- **Think about implementing classes for different types of vehicles**
  - Motor bikes, Sports cars, trucks
- **There are many aspects that are common between vehicles**
  - Components: Engines, wheels, brakes and etc.
  - Functions: accelerating, braking, steering and etc.
- **For now, we need to implement all the common aspects separately. → Redundancy in codes**
  - Same code everywhere.
- **Inheritance is mainly about grouping common aspects.**
  - Extract general features of classes

Example (cont.)



- **All cars are vehicles.**
  - Main components and functions are implemented in the Vehicles class.
- **Bikes, cars, buses and trucks inherits characters from the Vehicles class.**
  - It makes us not to implement those common aspects in **derived** classes.
  - Distinctive features of each derived classes are implemented separately.
- **Same thing happens with lower levels.**

Base Class vs. Derived Class

- **Base class: parent class or superclass**
  - **General** class from which others can derive
  - It can be used as is.
- **Derived class: child class or subclass**
  - A new class inherited from base class
  - **Automatically has** member variables/functions of a base class.
  - We can **add additional** member functions and variables.
- **Similar to simulate family relationships**
  - Ancestor class: a parent of a parent …
  - Descendant class: opposite of ancestor

EXAMPLE: EMPLOYEE CLASSES

General Class of Employees

- **Class of Employees**
  - Composed of: salaried employees and hourly employees
  - Each class is "**subset**" of employees

- **Considering general concepts of employee is helpful.**
  - All have names and social security numbers (SSN)
  - Associated functions for these "basics" are the same among all employees.

Employee Class

- **Many members of "employee" class apply to all other types of employees**
  - Accessor/mutator functions
  - Most data items: Name, SSN, Pay
  - However, we **will not** have "objects" of this class.
    - None are just employees but salaried or hourly.
    - Need to redefine different behaviors depending on employee types.
- **Consider printCheck() function:**
  - Will always be "redefined" in derived classes so that different employee types can have different checks.
  - Makes no sense for "undifferentiated“ employees
    - Needs to show an error when is called in Employee class

Base Class Example: Employee

Display 14.1 Interface* for Base Class Employee

```
class Employee
{
public:
    Employee();
    Employee( const string& theName, const string& theSsn );
    string getName() const; // "const": no member modification allowed
    string getSsn() const;
    double getNetPay() const;
    void   setName(const string& newName);
    void   setSsn(const string& newSsn);
    void   setNetPay(double newNetPay);
    void   printCheck() const; // will be redefined in children

private:
     string name;
     string ssn;
     Double netPay;
};
```

*Interface(API): rules for how to use the class

Deriving from Employee Class

- **Derived classes from Employee class:**
  - Automatically have all member variables.
  - Automatically have all member functions.
  - So, a derived class is said to *inherit* members from the base class.

- **In derived classes,**
  - we can redefine existing members,
    - In our example, `printCheck()`
  - and/or add new members (variables and functions).

HourlyEmployee Class Interface/Additions

Syntax

- *class subclass_name : access_mode baseclass_name*

The heading:

- Specifies "publicly inherited" from Employee class

```
class HourlyEmployee : public Employee
{ …
```

Additions

- Derived class interface only lists new or "to be redefined" members
- Since all others inherited are already defined
- HourlyEmployee adds:
  - Constructors
  - wageRate, hours member variables
  - setRate(), getRate(), setHours(), getHours() member functions

Derived Class Example

Display 14.3 Interface for HourlyEmployee

- See next pages for explanation

```
class HourlyEmployee : public Employee
{
public:
    HourlyEmployee( );
    HourlyEmployee(const string& theName, const string& theSsn,
        double theWageRate, double theHours);
    void   setRate(double newWageRate);
    double getRate( ) const;
    void   setHours(double hoursWorked);
    double getHours( ) const;
    void   printCheck( ); // redefining printCheck() of Employee
private:
    double wageRate;
    double hours;
};
```

Among inherited member functions, list only the declarations of inherited ones that you want to change the definition of the function.

Person and student class


```
 1 #include <iostream>
 2 #include <string>
 3 using namespace std;
 4
 5 class Person
 6 {
 7   public:
 8     Person() : name("not set") {}
 9     Person(string name) : name(name) {}
10     string getName(string name) const {return name;}
11     void setName(string name) {this->name=name;}
12     void printInfo() const;
13   private:
14     string name;
15 };
16
17 void Person::printInfo() const
18 {
19   cout << "Name: " << name << endl;
20 }
21
22 class Student : public Person
23 {
24   public:
25     Student(string name) : Person(name) {}
26     void setSid(int sid) {this->sid = sid;}
27     int getSid() const {return sid;}
28   private:
29     int sid;
30 };
31
32 int main(){
33   Student st1("Jason");
34   st1.printInfo();
35
36   return 0;
37 }
```

Base Class

Derived Class

Summary

- Inheritance Concepts
- Employee : Base Class / Parent Class
- Hourly Employee : Derived Class / Child Class

REDEFINING MEMBER FUNCTIONS

Redefinition of Member Functions

Recall: interface of derived class:

- When inherited member functions are **NOT** declared, they are automatically inherited.

```
class Employee
{
    void   printCheck() const; // general printCheck()
```

Redefining

- We can change the behavior of inherited member functions.
- For this, they need to declare **explicitly** (with the **same signature**).
- This is called "**redefining**" (member functions of bases classes).
- C++ allows us to drop the `const` when redefining in the derived class.

```
class HourlyEmployee : public Employee
{
    void   printCheck(); // printCheck() for Hourly Employee
```

Redefining vs. Overloading

- They looks similar but are very different!

- Redefining in derived class:
  - SAME parameter list (signature)
  - Essentially "re-writes" same function

- Overloading:
  - Different parameter list (signature)
  - Defined "new" function that takes different parameters
  - Overloaded functions must have different signatures

Accessing Redefined (Original) Base Function

- Base class's definition not "lost" in derived class
- But, we can specify it's use explicitly:
  - Not typical, but useful sometimes

```
Employee       JaneE;
HourlyEmployee SallyH;
JaneE.printCheck(); // Employee's printCheck
SallyH.printCheck(); // HourlyEmployee printCheck
SallyH.Employee::printCheck(); // Employee's printCheck
```

Member function redefining


```
 1 #include <iostream>
 2 #include <string>
 3 using namespace std;
 4
 5 class Person
 6 {
 7   public:
 8     Person() : name("not set") {}
 9     Person(string name) : name(name)
       {}
10     string getName(string name) const
       {return name;}
11     void setName(string name)
       {this->name=name;}
12     void printInfo() const;
13   private:
14     string name;
15 };
16
17 void Person::printInfo() const
18 {
19   cout << "Name: " << name << endl;
20 }
21
```

```
22 class Student : public Person
23 {
24   public:
25     Student(string name) : Person(name)
       {}
26     void setSid(int sid)
       {this->sid = sid;}
27     int getSid() const {return sid;}
28     void printInfo();
       // const is dropped for demo
29   private:
30     int sid;
31 };
32
33 void Student::printInfo(){
34   Person::printInfo();
35   cout << "Student ID: " << sid << endl;

36 }
38 int main(){
39   Student st1("Jason");
40   st1.setSid(10);
41   st1.printInfo();
42   cout << "==========" << endl;
43   st1.Person::printInfo();
45   return 0;
46 }
```

Summary

- Redefinition of member functions
- Overloading vs redefining functions
- Accessing redefined original member functions

CONSTRUCTORS IN DERIVED CLASSES

Constructors in Derived Classes

- **Base class constructors are not inherited in derived classes.**
  - But, they **can be invoked** within derived class constructor.

- **Base class constructor must initialize all base class member variables.**
  - Those member variables are inherited by derived class.
  - So, the derived class constructor simply calls it to initialize them.
    - "First" thing derived class constructor does

Example: Derived Class Constructor

- Consider syntax for HourlyEmployee constructor:

```
HourlyEmployee::HourlyEmployee(string theName,
          string theNumber, double theWageRate,
          double theHours)
          : Employee(theName, theNumber),
          wageRate(theWageRate), hours(theHours)
{
   // deliberately empty
}
```

- Initialization section
  - Includes invocation of Employee constructor
  - Initializing the base class members in initialization section is not allowed.

Another HourlyEmployee Constructor

- A second constructor:

```
HourlyEmployee::HourlyEmployee()
      : Employee(), wageRate(0), hours(0) { // also empty }
```

  - Default version of base class constructor is called (no arguments)
  - Should always invoke one of the base class's constructors
  - If you do not, **default** base class constructor **automatically called**. Then, its equivalent is:

```
HourlyEmployee::HourlyEmployee()
      : wageRate(0), hours(0){ // also empty  }
```

Destructors in Derived Classes

- **When derived class destructor is invoked:**
  - Automatically calls base class destructor
  - So no need for explicit call

- **So derived class destructors need only be concerned with newly defined member variables of derived class.**
  - And any data they "point" to
  - Base class destructor handles inherited data automatically

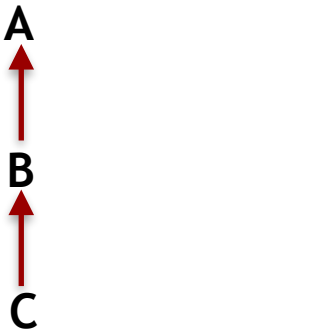
Constructor/Destructor Calling Order

Consider:

- class B derives from class A, class C derives from class B
- ctor is called when:
  - a local object of class C is created in a brace-scoped block
  - explicitly created using `new` or `new[]`
- dtor is called when:
  - object of class C goes out of scope
    - i.e., after function call or outside the braced-scope block
  - explicitly deleted using `delete` or `delete[]`

Calling order:

- ctor calling order: A→B→C
- dtor calling order: A←B←C



Inheritance Relationship

Inheritance with ctors and dtors


```
 1 #include <iostream>
 2 using namespace std;
 3
 4 class Person
 5 {
 6   public:
 7     Person() : name("not set") { cout << "Person: default constructor is
called" << endl; }
 8     Person(string name) : name(name) { cout << "Person(string): default
constructor is called" << endl; }
 9     string getName(string name) const {return name;}
10     void setName(string name) {this->name=name;}
11     void printInfo() const;
12   private:
13     string name;
14 };
15
```

```
16 class Student : public Person{
17   public:
18     Student() : Person(), sid(0) { cout << "Student: default constructor is
called" << endl; }
19     Student(int sid) : Person(), sid(sid) { cout << "Student(int):
constructor is called" << endl;}
20     Student(string name, int sid) : Person(name), sid(sid) { cout <<
"Person(string,int): constructor is called" << endl; }
21     void setSid(int sid) {this->sid = sid;}
22   private:
23     int sid;
24 };
25
26
27 int main(){
28   Student st1;
29   Student st2(10);
30   Student st3("Jack", 20);
31   return 0;
32 }
```

Please check when the default ctor of the base class is called.

PROTECTED QUALIFIER WITH INHERITANCE & OTHERS

Pitfall: Private Members in Base Class

- **Derived class "inherits" private members of bases classes**
  - But we still cannot directly access them.
  - Not even through in derived class member functions!
  - We may indirectly access them via accessor/ mutator member functions. (i.e., helper/wrapper functions)

```
class Person {
  …
  private:
    string name
};
class Student : public Person{
  …
  void printInfo();
};
void Student::printInfo(){
  cout << "Name: " << name << endl;  // NO!!
  cout << "Student ID: " << sid << endl;
}
```

Pitfall: Private Members in Base Class

- **However, we often need to access private members in Base Class.**
  - This is possible with protected qualifier (in base class).
  - In practice, `protected` is used more often then `private`.
    - Using `private` is rare in real applications.

`protected` Qualifier

- Allows access "by name" in derived class
  - In (derived) class, it acts like private
  - But nowhere else (i.e., not accessible outside of the class definitions)

```
class Employee {
    ...
protected:
    double wageRate;
    double hours;
};
class hourlyEmployee : public Employee { … };
```

- Considered "protected" in derived class
  - To allow future derivations in deeper derived classes (e.g., grandchildren)

`protected` and `private` Inheritance

New inheritance "forms"

- Both are rarely used
- The access scopes are reduced in derived classes

Protected inheritance:

```
class SalariedEmployee : protected Employee
{…}
```

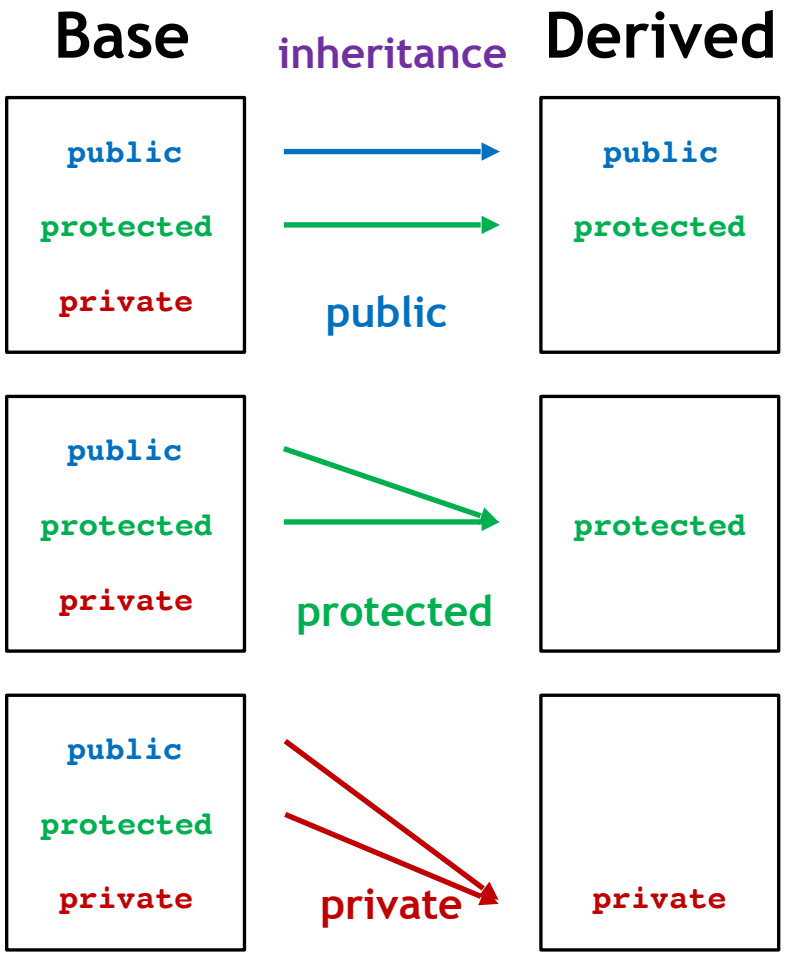
- `public` members in base class ➔ `protected` in derived class

Private inheritance:

```
class SalariedEmployee : private Employee
{…}
```

- `public` and `protected` in base class ➔ `private` in derived class

`protected` and `private` Inheritance



Functions Not Inherited

- **All normal functions in base class are inherited**
- **Exceptions include:**
  - Constructors
  - Destructors
  - Copy constructor
    - But if not defined, generates "default" one
    - Recall need to define one for pointers!
  - Assignment operator
    - If not defined → default

Copy Constructor Example

```
B::B(const B& Object)
             : A(Object), …
{…}
```

What is the type of `Object` in the above example?

- **Invocation of base copy constructor**
  - `": A (Object)"` in the code
  - Sets inherited member variables of derived class object being created
  - Note that `Object` is of type `B`
    - but it's also of type `B`, so argument is valid
    - The derived class type can be used as a base class type

Multiple Inheritance

- **Derived class can have more than one base class!**
  - Syntax just includes all base classes separated by commas:

```
class derivedMulti : public base1, base2
{…}
```

- **Possibilities for ambiguity are endless!**
  - Dangerous undertaking!
  - Some believe should never be used
  - Certainly, should only be used be experienced programmers!
  - So, it's not allowed in the successors of C++ (e.g., JAVA)