Inheritance

Computer Programming for Engineers (DSAF003-42)

Instructor:

Youngjoong Ko (nlp.skku.edu)

This Week

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

INHERITANCE BASICS

Introduction

- Object-oriented programming 객체지향
  - is a Powerful programming technique.
  - provides abstraction dimension called ***inheritance.*** 상속!

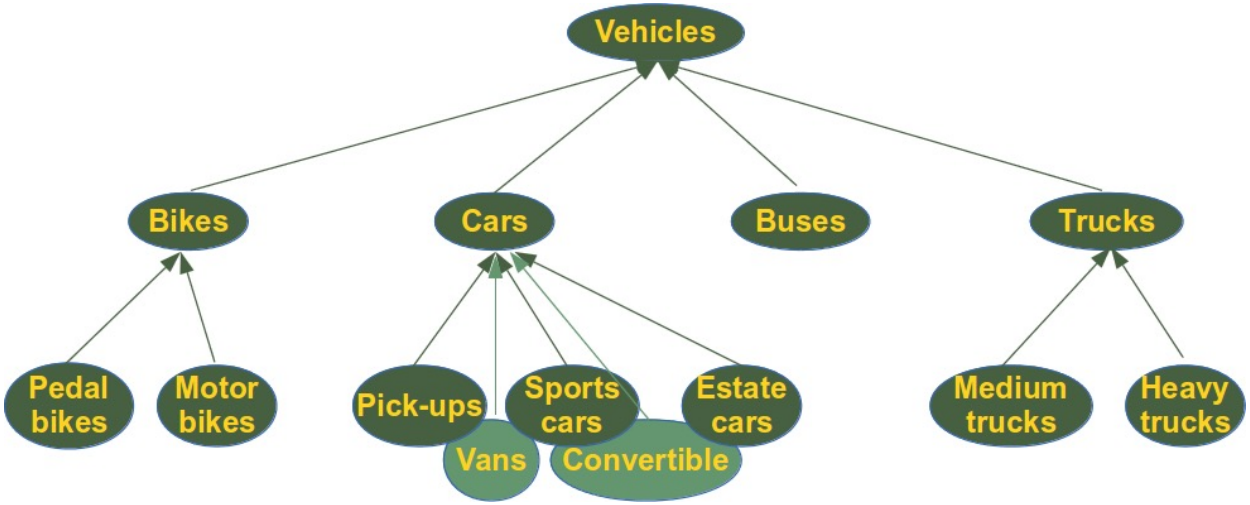
- How does it work?
  1. General form of class is defined
  2. Specialized versions then inherit properties of general class
  3. Can modify it's functionality for it's appropriate use

Example

- Think about implementing classes for different types of cars
  - Motor bikes, Sports cars, trucks
- There are many aspects that are common between cars
  - Components: Engines, wheels, brakes etc.
  - Functions: accelerating, braking, steering 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

공통적인 특징을 그룹화

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
- Terminology
  - Parent/child can be used in any tree structures but base/derive is only for inheritance. (So, more **precise**.)

EXAMPLE: EMPLOYEE CLASSES

General Class of Employees

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

  하위집합

- 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.

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

    ...
};
```

*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).

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;
};
```

HourlyEmployee Class Interface/Additions

- The heading:
  - Specifies "publicly inherited" from Employee class

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

- Additions

상속된 것이는 새 멤버 or 재정의된 멤버만!

  - 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

Person and student class


```
class Person
{
  public:
    Person() : name("not set") {}
    Person(string name) : name(name) {}
    string getName(string name) const {return name;}
    void setName(string name) {this->name=name;}
    void printInfo() const;
  private:
    string name;
};


void Person::printInfo() const
{
  cout << "Name: " << name << endl;
}

class Student : public Person
{
  public:
    void setSid(int sid) {this->sid = sid;}
    int getSid() const {return sid;}
  private:
    int sid;
};
```

Derived Class

REDEFINING MEMBER FUNCTIONS

Redefinition of Member Functions

- Recall: interface of derived class:
  - When inherited member functions are **NOT** declared, they are automatically inherited.

상속된 멤버함수 선언X
→ 자동으로 상속

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

- Redefining 재정의
  - 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.

const 생략 가능

```
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)** 같은 매개변수
  - Essentially "re-writes" same function 같은 기능을 다시 작성.

- Overloading:
  - **Different parameter list (signature)** 다른 매개변수
  - Defined "new" function that takes different parameters 새 함수!
  - 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

파생이나 기본이 접근이 사라지는 거 X!

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

접근 가능!

Member function redefining


```
class Person
{
  public:
    ...
    void printInfo() const;
  private:
    string name;
};

class Student : public Person
{
  public:
    ...
    void printInfo(); // const is dropped for demo
  private:
    int sid;
};

void Student::printInfo()
{
  Person::printInfo();
  cout << "Student ID: " << sid << endl;
}
```

이경우 Person의 printInfo와 같이 출력!

Redefined Member Function

CONSTRUCTORS IN DERIVED CLASSES

Constructors in Derived Classes

파생 클래스의 생성자

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

기본 클래스 생성자는 파생된 클래스에 상속 X.
but 호출 O

- 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

기본 클래스 생성자는 모든 멤버 초기화해야.

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 초기화 섹션
  - Includes invocation of Employee constructor
  - Initializing the base class members in initialization section is not allowed.
    기반 클래스의 멤버 초기화 가능

Another HourlyEmployee Constructor

- A second constructor:

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

↑ 기본 호출은 생략해도 됨

  - 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:

안 써주면 자동으로 default가 호출됨.

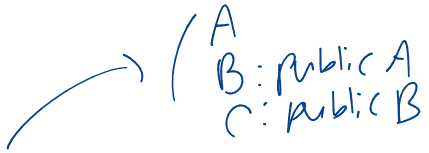
```
HourlyEmployee::HourlyEmployee()
        : wageRate(0), hours(0){ // also empty  }
```

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 with ctors and dtors (3_)


```
class Student : public Person
{
  public:
    Student() : Person(), sid(0) {}
    Student(int sid) : Person(), sid(sid) {}
    Student(string name, int sid) : Person(name), sid(sid) {}
    void setSid(int sid) {this->sid = sid;}
    int getSid() const {return sid;}
    void printInfo();
  private:
    int sid;
};
```

Redefined ctors

Student Kim("A", 0); ⇒ Person(name) ctor
Student(name, sid) ctor

PROTECTED QUALIFIER WITH INHERITANCE

Pitfall: Private Members in Base Class

기본 class의 private 멤버.

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

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

Pitfall: Private Members in Base Class

- However, we often need to access private members in BC.
  - 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 outside of class or derived class, It acts like private (i.e., not accessible at outside of the class definitions)

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

- 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