Chapter 14. 상속

박 종 혁 교수
UCS Lab
(http://www.parkjonghyuk.net)

Tel: 970-6702
Email: jhpark1@snut.ac.kr

Learning Objectives

- 상속의 기본
  - 파생 클래스와 생성자
  - protected: 제한자
  - 멤버 함수의 재정의
  - 상속되지 않는 함수들
- 상속을 이용한 프로그래밍
  - 할당 연산자와 복사 생성자
  - 파생 클래스에서의 소멸자
  - 다중 상속

상속 개요

- 객체 지향 프로그래밍
  - 강력한 프로그래밍 기법
  - 상속(inheritance)이라는 추상화를 위한 새로운 차원의 기법을 제공
- 일반화된 형태의 클래스가 먼저 정의
  - 일반화된 클래스를 상속받아 특성화된 버전이 정의됨
  - 이때, 상속받은 속성들을 적절하게 수정할 수 있음

상속의 기본

- 새로운 클래스는 다른 클래스로부터 상속이 가능
- 기반 클래스(Base Class)
  - 다른 클래스가 상속받는 일반 클래스
- 파생 클래스(Derived class)
  - 새로운 클래스
  - 자동으로 기반 클래스의 멤버 변수와 함수를 소유
  - 멤버 변수와 함수의 추가 가능

파생 클래스 (1/2)

- 예 : 직원(Employees) 클래스
- 다음으로 구성:
  - 봉급제 직원(Salaried employee)
  - 시간제 직원(Hourly employee)
- 각각은 직원의 부분집합
  - 전체 직원은 시간제 급여를 받는 직원들과 매주 또는 매달 고정 급여를 받는 직원들로 구성

파생 클래스 (2/2)

- 일반적인 직원(Employee)의 형이 아니다
  - 어느 누구도 단순하게 "직원"인 것은 아니다
- 직원의 일반화된 개념을 만들고 접근
  - 모든 직원은 이름을 가짐
  - 모든 직원은 사회 보장 번호 (주민등록번호)를 가짐
  - 이와 관련된 함수는 모든 직원에 적용
- 일반화된 클래스는 모든 직원이 공통적으로 공유하는 속성을 가짐

Employee 클래스 (1/2)

- Employee 클래스의 멤버 변수와 함수는 모는 형태의 직원에 적용
  - accessor 함수
  - mutator 함수
  - 데이터 속성:
    - SSN
    - Name
    - Pay
  - 이 클래스의 객체를 직접 사용하는 예는 없음

Employee 클래스 (2/2)

- printCheck() 함수 : (Display 14.2)
  - 항상 하위(파생) 클래스에서 재정의 됨
  - 직원 타입에 따라 다른 결과를 출력
  - "구분되지 않는" 직원을 위한 출력은 있을 수 없다
  - 따라서, Employee 클래스의 printCheck()
    - 이 부분이 출력 되어서는 안됨
    - 에러 메시지 출력, 시스템 종료 부분 추가

Employee 클래스로부터의 파생

- 파생 클래스:
  - 자동으로 모든 기반 클래스의 멤버 변수와 함수를 가짐
- 파생 클래스는 기반 클래스 멤버를 상속
- 멤버의 재정의와 새로운 멤버의 추가 가능

디스플레이 14.3 파생 클래스 HourlyEmployee의 인터페이스 (1/2)

Display 14.3 Interface for the Derived Class HourlyEmployee

```
1
2   //This is the header file hourlyemployee.h.
3   //This is the interface for the class HourlyEmployee.
4   #ifndef HOURLYEMPLOYEE_H
5   #define HOURLYEMPLOYEE_H

6   #include <string>
7   #include "employee.h"

8   using std::string;

9   namespace SavitchEmployees
10  {
```

디스플레이 14.3 파생 클래스 HourlyEmployee의 인터페이스 (2/2)

```
11      class HourlyEmployee : public Employee
12      {
13      public:
14          HourlyEmployee( );
15          HourlyEmployee(string theName, string theSsn,
16                                  double theWageRate, double theHours);
17          void setRate(double newWageRate);
18          double getRate( ) const;
19          void setHours(double hoursWorked);
20          double getHours( ) const;
21          void printCheck( ) ;
22      private:
23          double wageRate;
24          double hours;
25      };

26  }//SavitchEmployees

27  #endif //HOURLYEMPLOYEE_H
```

You only list the declaration of an inherited member function if you want to change the definition of the function.

HourlyEmployee 클래스 인터페이스

- 다른 클래스와 유사함
  - #ifndef 구조
  - 필요 라이브러리 include
  - include employee.h!
- 클래스 헤더 부:

  class HourlyEmployee : public Employee

  { …
  - Employee 클래스로부터 public 상속을 받았음을 명시

HourlyEmployee 클래스 추가사항

- 파생 클래스의 인터페이스에서는 새로운 멤버를 추가하거나 재정의 하는 부분만 추가됨
  - 모든 상속 받은 부분은 정의되어 있음
  - ssn, name, …
- HourlyEmployee 추가 부분:
  - 생성자
  - 멤버 변수: wageRate, hours
  - 멤버 함수: setRate(), getRate(), setHours(), getHours()

HourlyEmployee 클래스 재정의

- HourlyEmployee 재정의: (Display 14.5)
  - printCheck() 멤버 함수
  - Employee 클래스로부터 상속받은 printCheck() function 구현을 오버라이딩(overriding)
- 이 함수의 정의 부분은 HourlyEmployee 클래스의 구현부에 있어야 함

상속 (Inheritance) 용어

- 일반적으로 가족(부모와 자식 사이) 간의 관계를 표현
- 부모 클래스(parent class)
  - 기반 클래스
- 자식 클래스(child class)
  - 파생 클래스
- 조상 클래스(ancestor class)
  - 부모의 부모…
- 자손 클래스(descendant class)
  - 자식의 자식…

파생 클래스 생성자

- 기반 클래스의 생성자는 상속되지 않음!
  - 기반 클래스의 생성자는 파생클래스의 생성자 내에서 호출 가능 → 필요함!
- 기반 클래스 생성자는 기반 클래스 멤버 변수를 초기화 해야 함
  - 따라서 파생 클래스 생성자에서 호출하여 기반 클래스 멤버를 초기화
    - 첫 번째 파생 클래스 생성자

파생 클래스 생성자 예

- HourlyEmployee 생성자:

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

- 초기화 섹션에 Employee 클래스의 생성자 호출

HourlyEmployee의 다른 생성자

- 두 번째 생성자:

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

- 매개변수가 없는 디폴트 버전의 기반 클래스 생성자 호출
- 항상 기반 클래스의 생성자 중 하나를 호출 해야 함

기반 클래스 호출이 없는 생성자

- 파생 클래스의 생성자는 항상 기반 클래스의 생성자 중 하나를 호출 해야 함
- 하지 않으면:
  - 기반 클래스의 디폴트 생성자가 자동으로 호출
- 다음 버전에서 자동으로 호출:

```
HourlyEmployee::HourlyEmployee()
                        : wageRate(0), hours(0)
{ }
```

함정: 기반 클래스 private 멤버 변수

- 파생 클래스는 private 멤버 변수를 상속 함
  - 하지만, 직접적인 접근은 불가능
  - 파생 클래스의 멤버 함수에서도 불가능
- private 멤버 변수는 자신들이 정의된 클래스의 멤버 함수를 통해서만 접근이 가능

함정: 기반 클래스 private 멤버 함수

- 기반 클래스의 private 멤버 함수도 동일한 규칙이 적용
  - 기반 클래스의 인터페이스 및 구현부 외부에서의 접근이 불가능
  - 파생 클래스 멤버 함수에서도 불가능

함정: 기반 클래스 private 멤버 함수 영향

- private 멤버 변수
  - 기반 클래스의 멤버 함수를 통하여 간접적인 접근이 가능
- private 멤버 함수
  - 접근이 불가능
- 합리적 이유
  - private 멤버 함수는 helper 함수
  - 단지 정의된 곳에서만 사용됨

protected 제한자

- 클래스 멤버의 새로운 분류
- 파생 클래스에서 직접 접근이 가능
  - 다른 클래스에서 접근이 불가
- private 처럼 동작한다
- 미래의 자손들에게 접근을 허용
- 일부 전문가들은 정보 은닉의 규칙을 파괴했다고 생각함

멤버 함수의 재정의

- 파생 클래스의 인터페이스:
  - 새로운 멤버 함수 선언
  - 또한, 상속 받은 함수 중 변경하고자 하는 함수를 선언
  - 상속받은 함수 중 선언을 하지 않은 함수
    - 변경하지 않고 사용
- 파생 클래스 구현 파일:
  - 새로운 멤버 함수 정의
  - 재정의 하고자 하는 기반 클래스의 멤버 함수 정의도 포함되어야 함

재정의 (Redefining) vs. 오버로딩 (Overloading)

- 매우 다름!
- 파생 클래스에서 재정의 됨(상속관계에서):
  - 같은 매개변수 리스트를 가짐
  - 같은 함수의 재작성이 필요
- 오버로딩:
  - 다른 매개변수 리스트
  - 새로이 정의된 함수는 다른 매개변수를 가짐
  - 오버로딩된 함수는 다른 시그니처를 가짐

함수 시그니처

- 함수 시그니처:
  - 함수 이름
  - 매개변수 리스트
    - 순서, 개수, 타입
- 시그니처가 포함하지 않는 것:
  - 리턴 타입
  - const 키워드
  - &

재정의된 기반 클래스 함수에 대한 접근

- 파생 클래스에서 재정의 되었다고 해서 기반 클래스의 함수를 상실하는 것은 아님
- 재정의된 기반 클래스 함수 사용:

```
Employee     JaneE;
HourlyEmployee SallyH;
JaneE.printCheck();  → calls Employee's
                                printCheck function
SallyH.printCheck();  → calls HourlyEmployee
                                printCheck function
SallyH.Employee::printCheck();  → Calls Employee's
                                printCheck function!
```

- 때때로 기반 클래스의 함수가 사용됨

상속되지 않는 함수들

- 예외:
  - 생성자
  - 소멸자
  - 복사 생성자
    - 정의 하지 않으면 → 디폴트 버전이 생성됨
    - 포인터 변수를 사용하여 동적할당을 실행할 경우 → 정의해야 함
  - 할당 연산자
    - 정의 하지 않으면 → 디폴트 버전

할당연산자와 복사 생성자

- 오버로딩된 할당 연산자와 복사 생성자는 상속되지 않음
  - 하지만 파생 클래스의 정의 부분에서 사용 가능
  - 일반적으로 반드시 사용하여 재정의 해야 함
  - 파생 클래스의 생성자에서 기반 클래스의 생성자를 사용하는 것과 유사

할당 연산자 예

- 파생 클래스 코드:

```
Derived& Derived::operator =(const Derived &
rightSide)
{
    Base::operator =(rightSide);
    …
}
```

  - 기반 클래스의 연산자 호출
    - 상속된 멤버를 처리

복사 생성자 예

- 파생 클래스 코드:

```
Derived::Derived(const Derived& Object)
                                : Base(Object), …
{…}
```

- 기반 클래스의 복사 생성자 호출
  - 상속된 멤버 변수를 처리
  - 기반 클래스의 생성자에 넘겨주는 객체는 파생 형 → 파생 형도 기반 클래스 형이므로 매개변수로 넘겨주는 것이 가능

파생 클래스에서의 소멸자

- 기반 클래스의 소멸자가 완벽하게 정의 되어 있다면
  - 파생 클래스의 소멸자 정의는 쉬움
- 파생 클래스의 소멸자가 호출되면:
  - 자동으로 기반 클래스의 소멸자가 호출
  - 명시적 호출이 필요 없음
- 따라서, 파생 클래스의 소멸자는 파생클래스에서 정의된 변수에 대해서만 delete 수행

소멸자 호출 순서

- 다음을 고려:
  클래스 B는 클래스 A에서 파생
  클래스 C는 클래스 B에서 파생
  A ← B ← C
- 클래스 C의 객체가 소멸되려면:
  - 클래스 C의 소멸자 호출
  - 클래스 B의 소멸자 호출
  - 클래스 A의 소멸자 호출
- 생성자 호출 순서의 역순

"Is a" vs. "Has a" 관계

- 상속
  - "Is a" 클래스 관계
  - An HourlyEmployee "is a" Employee
  - A Convertible "is a" Automobile
- 하나의 클래스가 다른 클래스의 객체를 멤버로 가지고 있을 경우
  - "Has a" 클래스 관계
  - One class "has a" object of another class as it’s data

protected와 private 상속

- 새로운 상속의 형태
  - 잘 사용되지 않음
- protected 상속:
  class SalariedEmployee : protected Employee
  {…}
  - 기반 클래스의 public 멤버는 파생 클래스에서 protected 멤버로 변경됨
- private 상속:
  class SalariedEmployee : private Employee
  {…}
  - 기반 클래스의 모든 멤버는 파생 클래스에서 private 멤버로 변경됨
- Display 14.14 : 상속의 종류

다중 상속

- 파생 클래스는 하나 이상의 기반 클래스를 가질 수 있음!
  - 구문:
    class derivedMulti : public base1, base2
    {…}
- 혼동의 소지 있음!
  - 2개의 기반 클래스가 동일한 이름과 매개변수 자료형을 갖는 함수를 가지고 있다면?
- 위험한 작업!
  - 일부 전문가는 절대 사용해서는 안 된다고 주장
  - 충분히 경험을 쌓은 프로그래머가 사용해야 함!

요약 1

- 상속은 코드의 재사용을 제공
  - 파생 클래스에 몇 가지 특성만을 추가하여 사용
- 파생 클래스 객체는 기반 클래스의 멤버를 상속
  - 멤버 추가
- 파생클래스에서 기반 클래스의 private 멤버 변수는 직접 접근이 불가
- private 멤버 함수는 상속되지 않음

요약 2

- 상속된 멤버 함수의 재정의 가능
  - 기반 클래스에서의 동작과 다르게 정의 가능
- 기반 클래스의 protected 멤버:
  - 파생 클래스에서 직접 접근이 가능
- 오버로딩된 할당 연산자는 상속되지 않음
  - 파생클래스에서 호출 가능 → 파생 클래스 할당 연산자 정의에 사용 가능
- 생성자는 상속되지 않음
  - 파생클래스에서 호출 가능 → 파생 클래스 생성자 정의에 사용 가능

Q&A