

C++ Programming

9th Study: Object-Oriented Programming (5/8)

- Inheritance
- Method overriding
- Something about superclasses
- Up / down casting

C++ Korea 옥찬호 (utilForever@gmail.com)



Inheritance

When an object or class is based on another object or class, using the same implementation specifying implementation to maintain the same behavior

Inheritance

- 상속이란?
 - 일정한 친족 관계가 있는 사람 사이에서, 한 사람이 사망한 후에 다른 사람에게 재산에 관한 권리와 의무의 일체를 이어 주거나, 다른 사람이 사망한 사람으로부터 권리와 의무의 일체를 이어받는 일
- C++에서 상속이란, Is-A 관계!
 - Is-A 관계는 “무엇은 무엇의 한 종류이다.”라는 뜻
 - 서브클래싱, 확장, 상속이라는 용어들 모두 Is-A 관계를 말함
 - 예를 들어, 펭귄은 동물의 한 종류이다
→ 펭귄은 동물이 가지는 모든 특징을 가짐 + 펭귄만의 특징이 더해짐

Inheritance

- 클래스의 상속

- 클래스를 정의할 때 이미 존재하는 다른 클래스를 상속받을 수 있음
- 상속을 하면 새로 작성하는 클래스가 기존 클래스의 멤버 변수와 멤버 함수들을 자동으로 포함하게 됨
- 이 때, 기존 클래스를 부모 클래스 또는 슈퍼클래스라고 하고 새로 만들어지는 클래스를 자식 클래스 또는 서브클래스라고 함

Inheritance

- 클래스의 상속

- C++에서 클래스를 확장할 때는 확장할 클래스를 명시적으로 지정함
- 기존 클래스 Super와 새로 만들어지는 클래스 Sub가 있을 때, Sub 클래스를 Super 클래스로부터 상속해 만들려면 Sub 클래스를 정의할 때 Super 클래스를 지정

```
class Super
{
private:
    int mPrivateInt;
protected:
    int mProtectedInt;
public:
    Super();
    void someMethod();
};
```

```
class Sub : public Super
{
public:
    Sub();
    void someOtherMethod();
};
```

Inheritance

- 상속을 사용자 입장에서 보자면...
 - Sub 클래스의 객체가 Sub 타입이기도 하면서 Super 타입이기도 함
 - Sub가 Super를 상속받았기 때문
 - 이 말은 Sub 객체를 통해서 Sub의 public 멤버 변수 / 함수뿐만 아니라 Super가 가진 public 멤버 변수 / 함수도 이용할 수 있다는 의미
 - Sub 객체를 통해 Super 멤버 변수 / 함수를 이용할 때...
 - Sub에서 Super의 멤버 변수 / 함수를 직접 정의한 것처럼 이용 가능
 - someMethod()는 Super에서 정의된 함수지만 Sub 객체의 함수인 것처럼 호출

```
Sub mySub;  
mySub.someMethod();  
mySub.someOtherMethod();
```

Inheritance

- 상속은 단방향으로만 일어남!
 - Sub 클래스 입장에서는 Super 클래스와의 관계가 분명하지만, Super 클래스 입장에서는 자신을 상속받은 클래스가 무엇인지 모름
 - 즉, Super를 통해서는 Sub의 public 멤버 변수 / 함수에 접근 X
 - Sub는 Super일 수 있지만(Is-A 관계), Super는 Sub가 아님

```
Super mySuper;  
// 오류! Super에는 someOtherMethod()가 정의되어 있지 않음  
mySuper.someOtherMethod();
```

Inheritance

- 상속을 서브클래스 입장에서 보자면...
 - 일반 클래스와 마찬가지로 멤버 변수와 함수를 정의할 수 있음
 - 차이점이라면, 서브클래스에서는 슈퍼클래스에서 정의된 public, protected 멤버 변수 / 함수를 자신의 것처럼 이용할 수 있음

```
void Sub::someOtherMethod()  
{  
    std::cout << "mProtectedInt = " << mProtectedInt << std::endl;  
}
```

- 다시 한 번 정리해보는 private와 protected의 차이점
 - protected 멤버 변수 / 함수 : 서브클래스에서 접근 가능
 - private 멤버 변수 / 함수 : 서브클래스에서 접근 불가능

Method overriding

Allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes

Method overriding

- 클래스를 상속받는 이유는 기능을 추가하거나 바꾸기 위해서!
 - 서브클래스 Sub에서는 슈퍼클래스 Super에 새로운 멤버 함수 `someOtherMethod()`를 추가
 - 다른 멤버 함수 `someMethod()`는 Super로부터 상속받았으며, 당연하게도 슈퍼클래스에서와 완벽히 같게 동작함
 - 하지만, 많은 경우 상속을 받으면서 슈퍼클래스에서 이미 정의된 멤버 함수의 행동을 바꿔야 할 때가 있음
→ 메서드 오버라이딩(Method Overriding)!

Method overriding

- 메서드 오버라이딩과 virtual 속성
 - C++은 메서드 오버라이딩 특성을 제어하는 virtual 키워드를 제공
 - 메서드 오버라이딩이 동작하려면 슈퍼클래스에서 해당 멤버 함수가 virtual로 선언되어 있어야 함

```
class Super
{
private:
    int mPrivateInt;
protected:
    int mProtectedInt;
public:
    Super();
    virtual void someMethod();
};
```

Method overriding

- 메서드 오버라이딩과 virtual 속성
 - 멤버 함수가 오버라이딩이 될 지 안 될 지 정하는데 혼란이 생긴다면, 피할 수 있는 가장 편한 방법은 모든 멤버 함수에 virtual을 붙이는 것! (하지만 성능에서 약간 손해를 보게 됨)
 - Sub 클래스가 부모 역할을 할 일이 없다고 하더라도, 멤버 함수 선언에 virtual 키워드를 넣어 두는 것이 안전함

```
class Sub : public Super
{
public:
    Sub();
    virtual void someOtherMethod();
};
```

Method overriding

- 메서드 오버라이딩 문법

- 메서드를 오버라이딩할 때는 슈퍼클래스의 멤버 함수를 서브클래스에서 똑같이 선언해주고, 서브클래스의 구현부에서 해당 멤버 함수를 새롭게 정의

- 예를 들어, Super 클래스의 someMethod()가 정의되어 있을 때...

```
void Super::someMethod()  
{  
    std::cout << "Super::someMethod()" << std::endl;  
}
```

- virtual 키워드는 클래스에서 멤버 함수 원형을 선언할 때만 사용하면 되고 구현부에서는 사용할 필요가 없음

Method overriding

- 메서드 오버라이딩 문법

- someMethod()를 Sub 클래스에서 오버라이딩하기 위해서는 Sub 클래스의 정의에서 오버라이딩할 멤버 함수를 선언해 줘야 함

```
class Sub : public Super
{
public:
    Sub();
    virtual void someMethod();
};
```

- 새로운 someMethod()의 구현부를 다음과 같이 정의함

```
void Sub::someMethod()
{
    std::cout << "Sub::someMethod()" << std::endl;
}
```

Method overriding

- 메서드 오버라이딩 문법
 - 멤버 함수나 소멸자가 한 번이라도 virtual로 선언되고 나면, 서브클래스에서 오버라이딩할 때 virtual 사용 여부와 관계없이 무조건 virtual이 적용됨
 - Sub 클래스의 someMethod()는 슈퍼클래스 Super에서 이미 virtual로 선언했기 때문에 자동으로 virtual이 적용됨

```
class Sub : public Super
{
public:
    Sub();
    virtual void someMethod();
};
```

=

```
class Sub : public Super
{
public:
    Sub();
    void someMethod();
};
```

Method overriding


- 오버라이딩 메서드를 사용자 입장에서 보자면...
 - someMethod()가 오버라이딩되더라도 호출 방법은 달라지지 않음
 - 단, someMethod()의 정의가 클래스에 따라 바뀌기 때문에 어떤 클래스의 객체에서 멤버 함수를 호출했느냐에 따라 달라짐

Super mySuper;
mySuper.someMethod(); ➡ Super::someMethod()

Sub mySub;
mySub.someMethod(); ➡ Sub::someMethod()

Method overriding

- 오버라이딩 메서드를 사용자 입장에서 보자면...
 - 포인터나 참조를 통해서 슈퍼클래스 및 서브클래스를 가리킬 수 있음
 - 이 때, 포인터나 참조가 어떤 타입인지와 관계없이 객체 스스로 자신을 생성한 클래스를 알기 때문에 virtual로 선언된 멤버 함수의 호출이 일어나면 가장 마지막 서브클래스에서 오버라이딩된 멤버 함수를 찾아서 호출되게 함

```
Sub mySub;  
Super& ref = mySub;       Sub::someMethod()  
ref.someMethod();
```

- someMethod()를 호출하는 참조형 변수 ref는 Super 클래스 타입이지만 Sub 타입 객체를 가리키고 있기 때문에 Sub::someMethod()가 호출됨

Method overriding

- 오버라이딩 메서드를 사용자 입장에서 보자면...
- 참고 : 포인터나 참조가 슈퍼클래스의 타입이면 실제 가리키는 객체가 서브클래스라 하더라도 서브클래스에만 정의된 멤버 변수 / 함수에는 접근할 수 없음

```
Sub mySub;  
Super& ref = mySub;  
ref.someOtherMethod(); // X  
ref.someMethod();      // O
```

- 객체 스스로 자신이 생성된 클래스를 기억하고서 자신을 가리키는 클래스 타입과 관계없이 메서드 오버라이딩을 반영하는 동작은 포인터나 참조형 변수에서만 발생
- 개체 간 타입 캐스팅이나 대입을 통해 객체가 변경되는 경우에는 원본 객체의 클래스 정보가 기억되지 않음

Something about superclasses

Parent's constructor / destructor
References of parent class

Something about superclasses

- 부모의 생성자
 - C++의 생성 순서
 - 클래스가 부모 클래스를 가졌으면 부모 클래스의 디폴트 생성자가 실행됨
 - Static이 아닌 클래스 멤버들이 선언 순서에 맞춰서 생성됨
 - 클래스의 생성자가 실행됨
 - 생성 순서는 재귀적으로 적용됨
 - 클래스가 조부모를 가졌다면 조부모는 부모에 앞서 생성됨
 - 더 앞선 부모가 더 있을 때에도 마찬가지로 적용됨

Something about superclasses

- 부모의 생성자

```
class Something
{
public:
    Something() { std::cout << "2"; }
}

class Parent
{
public:
    Parent() { std::cout << "1"; }
}
```

```
class Child : public Parent
{
protected:
    Something mDataMember;
public:
    Child() { std::cout << "3"; }
}

int main() {
    Child myChild; // 123
}
```

- myChild 객체가 생성되면 Parent의 생성자가 가장 먼저 호출되면서 “1”을 출력
- 다음으로 mDataMember가 초기화되면서 Something의 생성자가 호출되어 “2”가 출력
- 마지막으로 Child의 생성자가 호출되면서 “3”을 출력

Something about superclasses

- 부모의 생성자

- C++은 부모 클래스의 디폴트 생성자가 있을 때 자동으로 호출해줌
- 만약 부모 클래스의 디폴트 생성자의 호출을 원하지 않거나, 디폴트 생성자가 정의되지 않는 상황이라면 멤버 이니셜라이저에서 직접 부모 클래스의 생성자를 지정할 수 있음

```
class Super {  
public:  
    Super(int i);  
}
```

```
class Sub : public Super {  
public:  
    Sub();  
}
```

```
Sub::Sub() : Super(7)  
{  
    // ...  
}
```

```
Sub::Sub(int i) : Super(i)  
{  
    // ...  
}
```

Something about superclasses

- 부모의 소멸자
 - C++의 소멸 순서
 - 클래스의 소멸자가 호출됨
 - 데이터 멤버를 생성 순서와 반대순으로 삭제
 - 부모 클래스가 있다면 소멸자를 호출
 - 소멸 순서는 재귀적으로 적용됨
 - 중요한 점 : 소멸자를 virtual로 선언해야 함!
 - 이유는 예제를 살펴보면...

Something about superclasses

- 부모의 소멸자

```
class Something {  
public:  
    Something() { cout << "2"; }  
    virtual ~Something() { cout << "2"; }  
}  
  
class Parent {  
public:  
    Parent() { cout << "1"; }  
    virtual ~Parent() { cout << "1"; }  
}
```

```
class Child : public Parent {  
protected:  
    Something mDataMember;  
public:  
    Child() { cout << "3"; }  
    virtual ~Child() { cout << "3"; }  
}  
  
int main() {  
    Child child; // 123321  
}
```

- main 함수가 끝나면 Child의 소멸자가 가장 먼저 호출되면서 “3”을 출력
- 다음으로 mDataMember가 삭제되면서 Something의 소멸자가 호출되어 “2”가 출력
- 마지막으로 Parent의 소멸자가 호출되면서 “1”을 출력

Something about superclasses

- 부모의 소멸자
 - 하지만 다음과 같이 소멸자에 virtual 키워드를 제거하고 객체가 Super 타입의 포인터로서 delete된다면 소멸 순서가 어떻게 될까?

```
class Something {  
public:  
    Something() { cout << "2"; }  
    ~Something() { cout << "2"; }  
}
```

```
class Parent {  
public:  
    Parent() { cout << "1"; }  
    ~Parent() { cout << "1"; }  
}
```

```
class Child : public Parent {  
protected:  
    Something mDataMember;  
public:  
    Child() { cout << "3"; }  
    ~Child() { cout << "3"; }  
}
```

```
int main() {  
    Parent* ptr = new Child();  
    delete ptr;  
}
```

Something about superclasses

- 부모의 소멸자

- 실행 결과는 엉뚱하게도 “1231”

- ptr 변수가 delete될 때 Parent의 소멸자만 호출됨
 - 왜냐하면 소멸자가 virtual로 선언되지 않았기 때문
 - 결과적으로 Child의 소멸자는 호출되지 못했고 데이터 멤버도 삭제되지 못함

- 해결책 : Parent의 소멸자만 virtual로 만들면 됨

- 부모에서 virtual로 선언했기 때문에 서브클래스에서도 자동으로 virtual이 적용됨

`~Parent() { std::cout << "1"; }`  `virtual ~Parent() { std::cout << "1"; }`

- 하지만 불필요한 불안요소를 없애려면 모든 소멸자를 virtual로 선언하는 것이 바람직

Something about superclasses

- 부모 클래스의 참조
 - 서브클래스에서 멤버 함수를 오버라이딩하면 슈퍼클래스의 멤버 함수가 완전히 교체됨

```
class Animal {  
public:  
    void printDesc() { cout << "Name of animal is "; }  
};
```

```
class Penguin : public Animal {  
public:  
    void printDesc() {  
        printDesc();  
        std::cout << "penguin." << std::endl;  
    }  
};
```

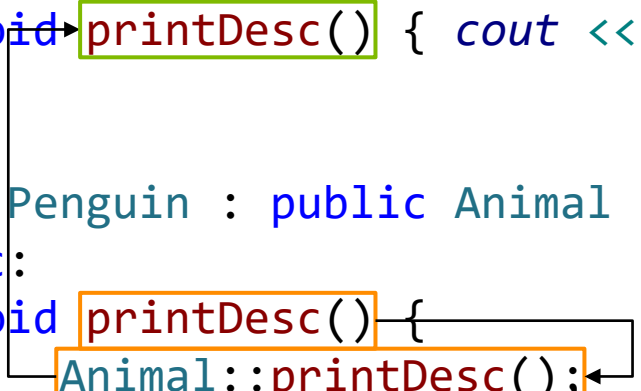
서브클래스에서 printDesc()를 오버라이딩
→ 슈퍼클래스의 printDesc()를 사용해
동물의 이름을 출력하려고 함

printDesc() = Penguin::printDesc()
→ 재귀적으로 무한히 반복
→ 스택 메모리 부족으로 인해 강제 종료

Something about superclasses

- 부모 클래스의 참조
 - 의도대로 동작하는 코드를 만들려면 스코프 지정 연산자를 사용

```
class Animal {  
public:  
    void printDesc() { cout << "Name of animal is "; }  
};  
  
class Penguin : public Animal {  
public:  
    void printDesc() {  
        Animal::printDesc();  
        std::cout << "penguin." << std::endl;  
    }  
};
```



Up / down casting

Converting a derived-class / base-class reference (pointer) to a base-class / derived-class

Up / down casting

- 업 캐스팅(Up-Casting) : 부모 클래스로 타입 캐스팅

```
Super mySuper = mySub;
```

- 객체를 대상으로 캐스팅이나 대입이 일어나면
자식 클래스의 특징들이 사라지는 슬라이싱(Slicing)이 발생
- 객체 간 대입으로 Sub 객체를 Super 객체로 업 캐스팅하면
Sub의 자식 클래스 특징들이 빠진 채 Super 객체의 값으로 갱신됨
- 하지만 참조형 변수의 초기화 방식으로 업 캐스팅을 하면
슬라이싱이 발생하지 않음 → 업 캐스팅의 올바른 방법!

```
Super& mySuper = mySub;
```

- 슬라이싱을 피하고 서브클래스의 객체를 그대로 사용할 수 있음

Up / down casting

- 다운 캐스팅(Down-Casting) : 자식 클래스로 타입 캐스팅
- 일반적으로 정상 동작이 보장되지 않기 때문에 거의 사용하지 않음

```
void presumptuous(Super* inSuper)
{
    Sub* mySub = static_cast<Sub*>(inSuper);
    mySub->someOtherMethod();
}
```

- presumptuous()의 내부를 알고 있다면 문제없이 사용할 수 있지만, 그렇지 않다면 큰 문제가 될 수 있음
→ 다운 캐스팅은 컴파일 시점에 검증이 안 되기 때문
- 아무런 근거도 없이 inSuper가 가리키는 포인터가 Sub의 객체일 거라고 가정해 버림

Up / down casting

- 다운 캐스팅(Down-Casting) : 자식 클래스로 타입 캐스팅
 - 다운 캐스팅을 꼭 써야만 한다면 `dynamic_cast`를 이용
 - `dynamic_cast`는 객체에 저장된 정보를 이용해 해당 캐스팅이 적합한지 런타임에 검사해 문제가 있으면 캐스팅을 거부하고 오류를 발생시킴
 - 포인터 변수에 대해 캐스팅이 실패했다면 결과 값으로 `nullptr`가 설정돼 무의미한 객체를 가리키게 됨

```
Sub* mySub = dynamic_cast<Sub*>(inSuper);  
if (mySub != nullptr)  
mySub->someOtherMethod();
```
 - 참조형 변수에 대해 캐스팅이 실패했다면 `std::bad_cast` 예외가 발생