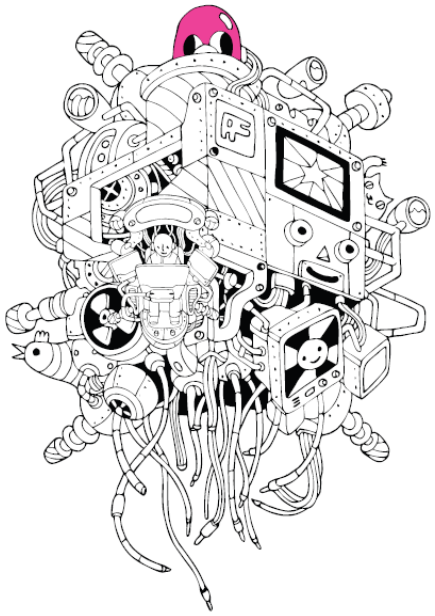
윤성우의
열혈 C++ 프로그래밍



윤성우 저 열혈강의 C++ 프로그래밍 개정판

Chapter 07. 상속의 이해

윤성우의

열혈 C++ 프로그래밍

Chapter 07-1. 상속에 들어가기에 앞서

윤성우 저 열혈강의 C++ 프로그래밍 개정판

상속의 이해를 위한 이 책의 접근방식

✔ 1단계: 문제의 제시

상속과 더불어 다형성의 개념을 적용해야만 해결 가능한 문제를 먼저 제시한다.

✔ 2단계: 기본개념 소개

상속의 문법적 요소를 하나씩 소개해 나간다. 그리고 그 과정에서 앞서 제시한 문제의 해결책을 함께 고민해 나간다.

✔ 3단계: 문제의 해결

처음 제시한 문제를, 상속을 적용하여 해결한다. 그리고 이 때 여러분의 감탄사를 기대한다.

위의 흐름대로 상속을 이해하기 바랍니다.

상속은 '기존에 정의해 놓은 클래스의 재활용을 목적으로 만들어진 문법적 요소'라고 이해하는 경우가 많다. 하지만 상속에는 다른, 더 중요한 의미가 담겨있다.

문제의 제시를 위한 시나리오의 도입


```
class PermanentWorker
{
private:
    char name[100];
    int salary;     // 매달 지불해야 하는 급여액
public:
    PermanentWorker(char* name, int money)
        : salary(money)
    {
        strcpy(this->name, name);
    }
    int GetPay() const
    {
        return salary;
    }
    void ShowSalaryInfo() const
    {
        cout<<"name: "<<name<<endl;
        cout<<"salary: "<<GetPay()<<endl<<endl;
    }
};
```

PermanentWorker는 정규직을 표현해 놓은 클래스이다.

```
class EmployeeHandler
{
private:
    PermanentWorker* empList[50];
    int empNum;
public:
    EmployeeHandler() : empNum(0)
    { }
    void AddEmployee(PermanentWorker* emp)
    {
        empList[empNum++]=emp;
    }
    void ShowAllSalaryInfo() const
    {
        for(int i=0; i<empNum; i++)
            empList[i]->ShowSalaryInfo();
    }
    void ShowTotalSalary() const
    {
        int sum=0;
        for(int i=0; i<empNum; i++)
            sum+=empList[i]->GetPay();
        cout<<"salary sum: "<<sum<<endl;
    }
    ~EmployeeHandler()
    {
        for(int i=0; i<empNum; i++)
            delete empList[i];
    }
};
```

프로그램 전체 기능의 처리를, 프로그램의 흐름을 담당하는 클래스를 가리켜 컨트롤 클래스라 한다.

EmployeeHandler의 경우 컨트롤 클래스에 해당한다.

신규 직원 등록 시

전체 급여정보 출력

급여 합계 정보 출력

문제의 제시

프로그램에 추가할 직급의 형태

- 영업직(Sales) 조금 특화된 형태의 고용직이다. 인센티브 개념이 도입
- 임시직(Temporary) 학생들을 대상으로 하는 임시고용 형태, 흔히 아르바이트라 함

확장 이후의 급여지급 방식

- 고용직 급여 연봉제! 따라서 매달의 급여가 정해져 있다.
- 영업직 급여 '기본급여 + 인센티브' 의 형태
- 임시직 급여 '시간당 급여 × 일한 시간' 의 형태

이 문제는 영업직과 임시직에 해당하는 클래스의 추가로 끝나지 않는다. 컨트롤 클래스인 EmployeeHandlder 클래스의 대대적인 변경으로 이어진다.

좋은 코드는 요구사항의 변경 및 기능의 추가에 따른 변경이 최소화되어야 한다. 그리고 이를 위한 해결책으로 상속이 사용된다.

윤성우의

열혈 C++ 프로그래밍

Chapter 07-2. 상속의 문법적인 이해

윤성우 저 열혈강의 C++ 프로그래밍 개정판

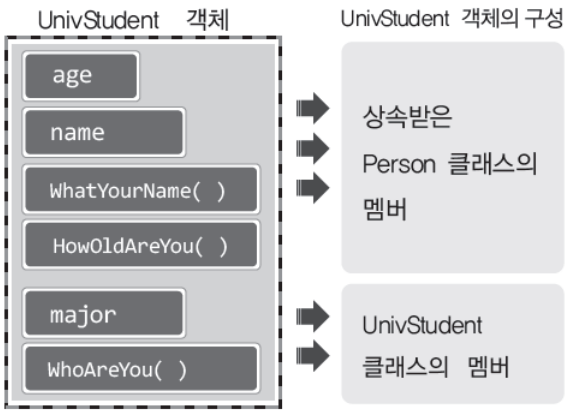
상속의 방법과 그 결과

```
class Person
{
private:
    int age;          // 나이
    char name[50];    // 이름
public:
    Person(int myage, char * myname) : age(myage)
    {
        strcpy(name, myname);
    }
    void WhatYourName() const
    {
        cout<<"My name is "<<name<<endl;
    }
    void HowOldAreYou() const
    {
        cout<<"I'm "<<age<<" years old"<<endl;
    }
};
```

```
class UnivStudent : public Person
{
private:
    char major[50];   // 전공과목
public:
    UnivStudent(char * myname, int myage, char * mymajor)
        : Person(myage, myname)
    {
        strcpy(major, mymajor);
    }
    void WhoAreYou() const
    {
        WhatYourName();
        HowOldAreYou();
        cout<<"My major is "<<major<<endl<<endl;
    }
};
```

Person 클래스를 public 상속함

Person 클래스의 멤버



Person	↔	UnivStudent
상위 클래스	↔	하위 클래스
기초(base) 클래스	↔	유도(derived) 클래스
슈퍼(super) 클래스	↔	서브(sub) 클래스
부모 클래스	↔	자식 클래스

용어정리

상속받은 클래스의 생성자 정의


```
UnivStudent(char * myname, int myage, char * mymajor)
    : Person(myage, myname)
{
    strcpy(major, mymajor);
}
```

이니셜라이저를 통해서 유도 클래스는 기초 클래스의 생성자를 명시적으로 호출해야 한다.

유도 클래스의 생성자는 기초 클래스의 멤버를 초기화 할 의무를 갖는다. 단! 기초 클래스의 생성자를 명시적으로 호출해서 초기화해야 한다.

```
int main(void)
{
    UnivStudent ustd1("Lee", 22, "Computer eng.");
    ustd1.WhoAreYou();

    UnivStudent ustd2("Yoon", 21, "Electronic eng.");
    ustd2.WhoAreYou();
    return 0;
};
```

때문에 유도 클래스 UnivStudent는 기초 크래스의 생성자 호출을 위한 인자까지 함께 전달받아야 한다.

private 멤버는 유도 클래스에서도 접근이 불가능하므로, 생성자의 호출을 통해서 기초 클래스의 멤버를 초기화해야 한다.

유도 클래스의 객체생성 과정


```
class SoBase
{
private:
    int baseNum;
public:
    SoBase() : baseNum(20)
    {
        cout<<"SoBase()"<<endl;
    }
    SoBase(int n) : baseNum(n)
    {
        cout<<"SoBase(int n)"<<endl;
    }
    void ShowBaseData()
    {
        cout<<baseNum<<endl;
    }
};
```

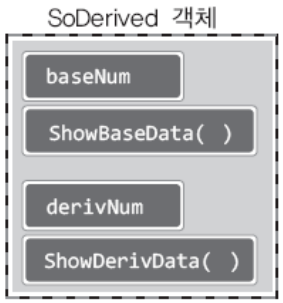
```
int main(void)
{
    cout<<"case1..... "<<endl;
    SoDerived dr1;
    dr1.ShowDerivData();
    cout<<"-------------------"<<endl;
    cout<<"case2..... "<<endl;
    SoDerived dr2(12);
    dr2.ShowDerivData();
    cout<<"-------------------"<<endl;
    cout<<"case3..... "<<endl;
    SoDerived dr3(23, 24);
    dr3.ShowDerivData();
    return 0;
};
```

```
class SoDerived : public SoBase
{
private:
    int derivNum;
public:
    SoDerived() : derivNum(30)
    {
        cout<<"SoDerived()"<<endl;
    }
    SoDerived(int n) : derivNum(n)
    {
        cout<<"SoDerived(int n)"<<endl;
    }
    SoDerived(int n1, int n2) : SoBase(n1), derivNum(n2)
    {
        cout<<"SoDerived(int n1, int n2)"<<endl;
    }
    void ShowDerivData()
    {
        ShowBaseData();
        cout<<derivNum<<endl;
    }
};
```

```
case1.....
SoBase()
SoDerived()
20
30
-------------------
case2.....
SoBase()
SoDerived(int n)
20
12
-------------------
case3.....
SoBase(int n)
SoDerived(int n1, int n2)
23
24
```

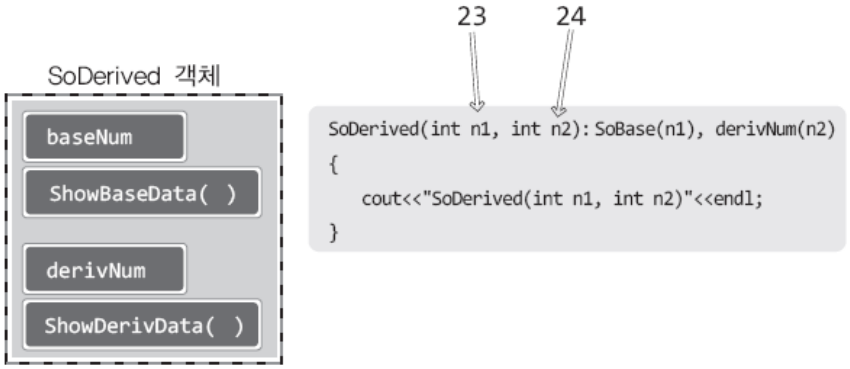
실행결과

유도 클래스의 객체생성 과정 case1

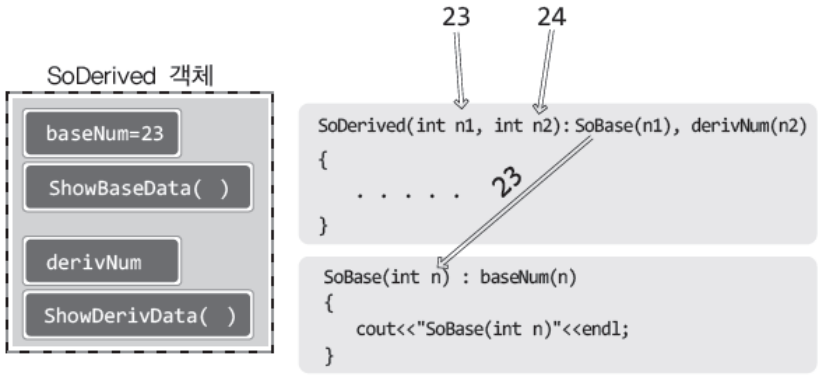


순서 1. 메모리 공간의 할당

```
SoDerived dr3(23, 24);
```



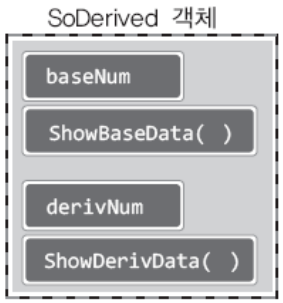
순서 2. 유도 클래스의 생성자 호출



순서 3. 기초 클래스의 생성자 호출 및 실행

순서 4. 유도 클래스의 생성자 실행

유도 클래스의 객체생성 과정 case2

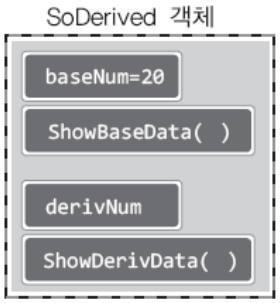


순서 1. 메모리 공간의 할당

SoDerived dr1


```
SoDerived( ) : derivNum(30)
{
    cout<<"SoDerived()"<<endl;
}
```

순서 2. 유도 클래스의 void 생성자 호출




```
SoDerived( ) : derivNum(30)
{
    cout<<"SoDerived()"<<endl;
}
```

순서 3. 이니셜라이저를 통한 기초 클래스의 생성자 호출이 명시적으로 정의되어 있지 않으므로 void 생성자 호출

```
SoBase( ) : baseNum(20)
{
   cout<<"SoBase()"<<endl;
}
```

순서 4. 유도 클래스의 실행

유도 클래스 객체의 소멸과정


```
class SoBase
{
private:
    int baseNum;
public:
    SoBase(int n) : baseNum(n)
    {
        cout<<"SoBase() : "<<baseNum<<endl;
    }
    ~SoBase()
    {
        cout<<"~SoBase() : "<<baseNum<<endl;
    }
};

class SoDerived : public SoBase
{
private:
    int derivNum;
public:
    SoDerived(int n) : SoBase(n), derivNum(n)
    {
        cout<<"SoDerived() : "<<derivNum<<endl;
    }
    ~SoDerived()
    {
        cout<<"~SoDerived() : "<<derivNum<<endl;
    }
};
```

```
SoBase() : 15
SoDerived() : 15
SoBase() : 27
SoDerived() : 27
~SoDerived() : 27
~SoBase() : 27
~SoDerived() : 15
~SoBase() : 15
```

실행결과

유도 클래스의 소멸자가 실행된 이후에 기초 클래스의 소멸자가 실행된다.

스택에 생성된 객체의 소멸순서는 생성순서와 반대이다.

유도 클래스 정의 모델


```
class Person
{
private:
    char * name;
public:
    Person(char * myname)
    {
        name=new char[strlen(myname)+1];
        strcpy(name, myname);
    }
    ~Person()
    {
        delete []name;
    }
    void WhatYourName() const
    {
        cout<<"My name is "<<name<<endl;
    }
};
```

```
class UnivStudent : public Person
{
private:
    char * major;
public:
    UnivStudent(char * myname, char * mymajor)
        : Person(myname)
    {
        major=new char[strlen(mymajor)+1];
        strcpy(major, mymajor);
    }
    ~UnivStudent()
    {
        delete []major;
    }
    void WhoAreYou() const
    {
        WhatYourName();
        cout<<"My major is "<<major<<endl<<endl;
    }
};
```

기초 클래스의 멤버 대상의 동적 할당은 기초 클래스의 생성자를 통해서, 소멸 역시 기초 클래스의 소멸자를 통해서

윤성우의

열혈 C++ 프로그래밍

Chapter 07-3. protected 선언과 세 가지 형태의 상속

윤성우 저 열혈강의 C++ 프로그래밍 개정판

protected로 선언된 멤버가 허용하는 접근의 범위

```
class Base
{
private:
    int num1;
protected:
    int num2;
public:
    int num3;
    void ShowData()
    {
        cout<<num1<<", "<<num2<<", "<<num3;
    }
};
```

```
class Derived : public Base
{
public:
    void ShowBaseMember()
    {
        cout<<num1;        // 컴파일 에러
        cout<<num2;        // 컴파일 OK!
        cout<<num3;        // 컴파일 OK!
    }
};
```

private < protected < public

private을 기준으로 보면,
protected는 private과 달리 상속관계에서의 접근을 허용한다!

세 가지 형태의 상속


```
class Derived : public Base
{
    . . . . .
}
```

public 상속

접근 제어 권한을 그대로 상속한다!

단, private은 접근불가로 상속한다!

```
class Derived : protected Base
{
    . . . . .
}
```

protected 상속

protected보다 접근의 범위가 넓은 멤버는 protected로 상속한다.

단, private은 접근불가로 상속한다!

```
class Derived : private Base
{
    . . . . .
}
```

private 상속

private보다 접근의 범위가 넓은 멤버는 protected로 상속한다.

단, private은 접근불가로 상속한다!

protected 상속과 private 상속


```
class Base
{
private:
    int num1;
protected:
    int num2;
public:
    int num3;
};
```

```
class Derived : private Base
{
    // empty!
};
```

private
상속의 결과

```
class Derived : private Base
{
접근불가:
    int num1;
private:
    int num2;
private:
    int num3;
};
```

때문에 이 이상의 상속은 무의미할 수 있다.

```
class Derived : protected Base
{
    // empty!
};
```

protected
상속의 결과

```
class Derived : protected Base
{
접근불가:
    int num1;
protected:
    int num2;
protected:
    int num3;
};
```

윤성우의

열혈 C++ 프로그래밍

Chapter 07-4. 상속을 위한 조건

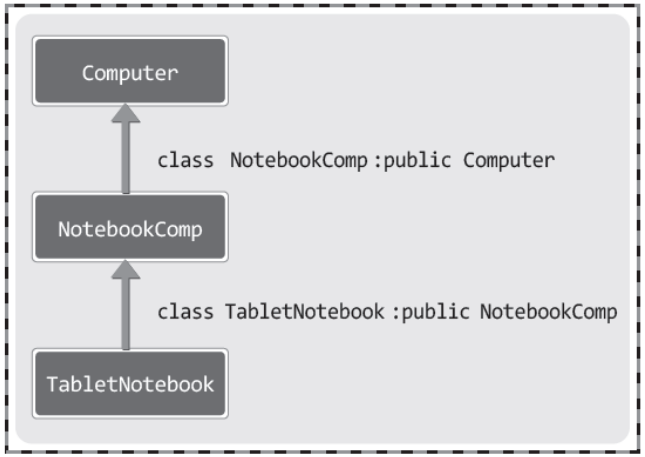
윤성우 저 열혈강의 C++ 프로그래밍 개정판

상속의 기본 조건인 IS-A 관계의 성립

- 전화기 → 무선 전화기
- 컴퓨터 → 노트북 컴퓨터

- 무선 전화기는 일종의 전화기입니다.
- 노트북 컴퓨터는 일종의 컴퓨터입니다.

- 무선 전화기 is a 전화기
- 노트북 컴퓨터 is a 컴퓨터

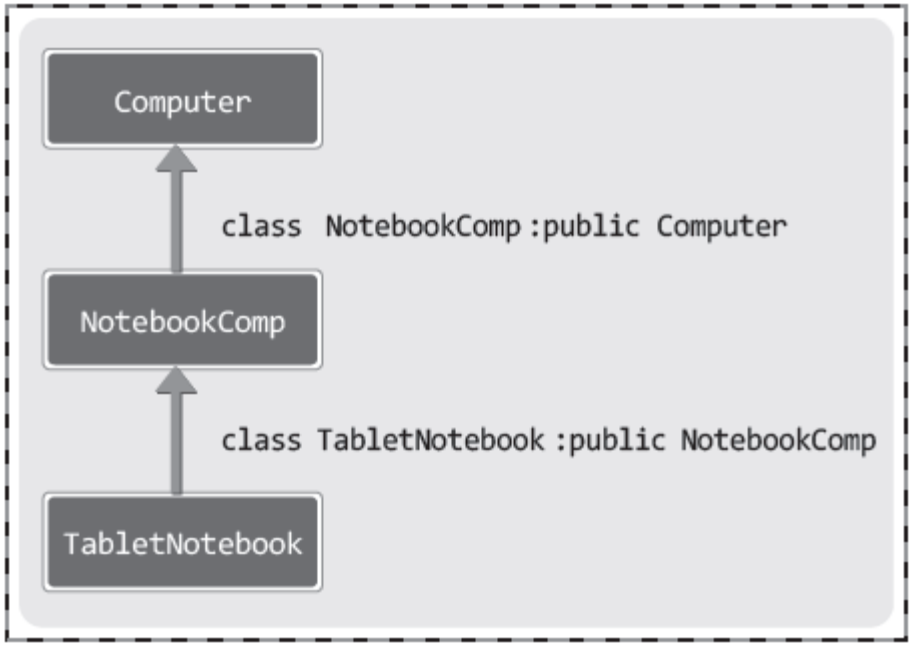


무선 전화기는 전화기의 기본 기능에 새로운 특성이 추가된 것이다.

노트북 컴퓨터는 컴퓨터의 기본 기능에 새로운 특성이 추가된 것이다.

이렇듯 is-a 관계는 논리적으로 상속을 기반으로 표현하기에 매우 적절하다.

IS-A 기반의 예제



예제는 도서 본문을 참조합시다!

- NotebookComp(노트북 컴퓨터)는 Computer(컴퓨터)이다.
- TabletNotebook(타블렛 컴퓨터)는 NotebookComp(노트북 컴퓨터)이다.
- TabletNotebook(타블렛 컴퓨터)는 Computer(컴퓨터)이다.

HAS-A 관계를 상속으로 구성하면


```
class Gun
{
private:
    int bullet;     // 장전된 총알의 수
public:
    Gun(int bnum) : bullet(bnum)
    { }
    void Shut()
    {
        cout<<"BBANG!";
        bullet--;
    }
};
```

```
class Police : public Gun
{
private:
    int handcuffs;      // 소유한 수갑의 수
public:
    Police(int bnum, int bcuff)
        : Gun(bnum), handcuffs(bcuff)
    { }
    void PutHandcuff()
    {
        cout<<"SNAP!";
        handcuffs--;
    }
};
```

경찰은 총을 소유한다.
경찰 has a 총!

has a 관계도 상속으로 구현이 가능하다. 하지만 이러한 경우 Police와 Gun은 강한 연관성을 띠게 된다.
따라서 총을 소유하지 않은 경찰이나, 다른 무기를 소유하는 경찰을 표현하기가 쉽지 않아진다.

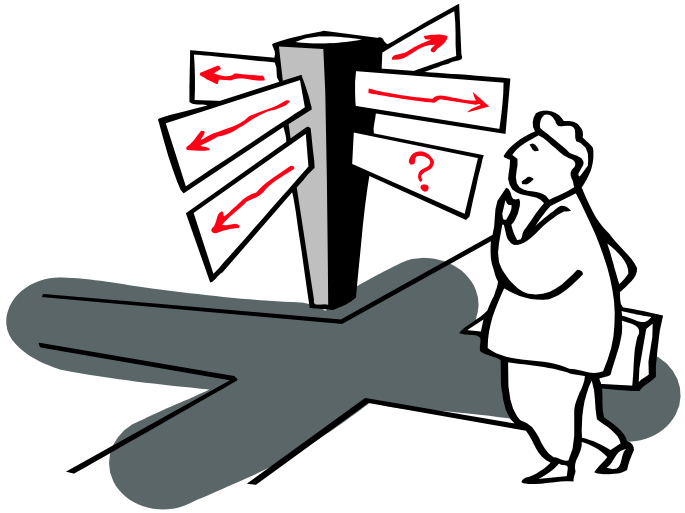
HAS-A 관계는 포함으로 표현한다


```
class Gun
{
private:
    int bullet;     // 장전된 총알의 수
public:
    Gun(int bnum) : bullet(bnum)
    { }
    void Shut()
    {
        cout<<"BBANG!"<<endl;
        bullet--;
    }
};
```

```
class Police
{
private:
    int handcuffs;      // 소유한 수갑의 수
    Gun * pistol;       // 소유하고 있는 권총
public:
    Police(int bnum, int bcuff)
        : handcuffs(bcuff)
    {
        if(bnum>0)
            pistol=new Gun(bnum);
        else
            pistol=NULL;
    }
    void PutHandcuff()
    {
        cout<<"SNAP!"<<endl;
        handcuffs--;
    }
    void Shut()
    {
        if(pistol==NULL)
            cout<<"Hut BBANG!"<<endl;
        else
            pistol->Shut();
    }
    ~Police()
    {
        if(pistol!=NULL)
            delete pistol;
    }
};
```

has a의 과계를 포함의 형태로 표현하면, 두 클래스간 연관성은 낮아지며, 변경 및 확장이 용이해진다.

즉, 총을 소유하지 않은 경찰의 표현이 쉬워지고, 추가로 무기를 소유하는 형태로의 확장도 간단해진다.



Chapter 07이 끝났습니다. 질문 있으신지요?