객체 지향 프로그래밍
(Object Oriented Programming)

13장

강사 – 강대기

차례 (Agenda)

- ISA 상속
- public으로 파생된 클래스
- protected 접근
- 생성자 멤버 초기자 리스트
- 업캐스트와 다운캐스트
- 가상 멤버 함수
- 초기 (정적) 결합과 말기 (동적) 결합
- 추상화 기초 클래스
- 순수 가상 함수 (pure virtual function)
- public 상속은 언제 어떻게?

상속으로 할 수 있는 일 (800쪽)

- 근본적으로 더 높은 수준의 재활용성 제공
- 기존의 클래스에 기능을 추가
- 클래스 안의 데이터에 다른 데이터 멤버 추가
- 클래스 메쏘드가 동작하는 방식을 변경

간단한 기초 클래스 (801~804쪽)

- TableTennisPlayer 클래스 파생 (803쪽)

```
class RatedPlayer : public TableTennisPlayer
{
}
```

- Public 상속 – 기초 클래스의 private
  - 기초 클래스의 public → 파생 클래스의 public
  - 기초 클래스의 private → 파생 클래스의 일부이나 접근 불가
- 파생 클래스는 (그림 13.1 in 804 쪽)
  - 기초 클래스의 데이터 멤버들을 저장
  - 기초 클래스의 멤버 함수들을 사용

상속 관계를 나타내는 도표 (protected는 844쪽 참조)

상속형태 / 기초 클래스	Public 상속	Protected 상속	Private 상속
Public 멤버	Public	Protected	Private
Protected 멤버	Protected	Protected	Private
Private 멤버	접근 불가	접근 불가	접근 불가

파생 클래스

- 초기자 리스트 (member initialization list) – 805쪽

```
RatedPlayer::RatedPlayer (unsigned int r, const char* fn, const char* ln, bool ht) :
    TableTennisPlayer (fn,ln,ht)
{
  this->rating = r;
}
RatedPlayer::RatedPlayer (unsigned int r, const char* fn, const char* ln, bool ht) :
    TableTennisPlayer (fn,ln,ht), rating(r)
{
}
```

- 객체 생성과 파괴 순서
  - 기초 클래스 생성자 → 파생 클래스 생성자
  - 초기자 리스트가 없으면 기초 클래스의 디폴트 생성자
  - 파생 클래스 파괴자 → 기초 클래스 파괴자

멤버 초기자 리스트

```
#include <iostream>
using namespace std;
class Test
{
private: int m_first; int m_second; // 순서가 중요
public:
  Test(int a) : m_first(a),m_second(m_first*2) { }
  friend ostream& operator<<(ostream& os, const Test &t)
    { return os << "first=" << t.m_first << ", second=" << t.m_second;          }
};
void main()
{
     Test t(4);
     cout << t << endl;
}
```

- 초기화 리스트의 초기식들은 리스트에 나타난 순서가 아니라 멤버의 선언 순서대로 실행
- m_first와 m_second 의 위치를 바꾸면 부작용이 발생함

Public 상속인 경우 – 파생 클래스와 기초 클래스와의 관계 (812쪽)

- 기초 클래스의 private 아닌 멤버를 사용
- 기초 클래스 포인터는 파생 클래스 지시 가능
- 기초 클래스 참조는 파생 클래스 참조 가능
- 기초 클래스 포인터나 참조는 기초 클래스 메쏘드만 호출 가능
- 파생 클래스 참조나 포인터는 기초 클래스를 지칭할 수 없음
- 기초 클래스 참조와 포인터를 전달인자로 사용하는 함수 → 파생 클래스 객체도 사용 가능
- 기초 클래스 객체를 파생 클래스 객체로 초기화 할 수 있음
- 파생 클래스 객체를 기초 클래스에 대입 가능

상속 (is-a 관계), 컴포지션 (has-a 관계)

- 814 쪽
- 바나나는 과일이지만(is-a), 점심은 바나나를 가지고 있다(has-a).
- Lunch 클래스의 한 데이터 멤버로 Fruit 객체를 포함시킨다.
- Array 클래스로부터 Stack 클래스를 파생시키지 말고, Stack 클래스 안에 Array 객체를 넣는다. (is-implemented-as-a)

다형성 – C++

- parametric polymorphism – 템플리트
- subtype polymorphism – 가상 함수 , 오버라이딩
- ad-hoc polymorphism – 오버로딩

public 다형 상속 (816쪽)

- 호출하는 객체마다 메쏘드의 행동이 다른 것
  - 오버라이딩 (subtype polymorphism)
- 방법
  - 기초 클래스 메쏘드를 다시 정의 (overriding) – 820쪽 코드
  - 가상 메쏘드(가상 함수)를 사용 – 821쪽 코드
- 가상 파괴자 – 올바른 순서로 파괴자가 호출되게 함 (833,840 쪽)

어떻게 상위 클래스 객체의 메쏘드를 실행하는가? (826쪽)

- 기초 클래스 메쏘드를 재정의 한 경우
  - Brass::ViewAcct(); // 기초 부분을 출력 (826쪽)
- 기초 클래스 메쏘드를 재정의 하지 않은 경우
  - Balance(); // (827쪽)
- Java – super.viewAcct();

단순 대입

```
Brass Piggy("Porcelot Pigg", 381299, 4000.00);
BrassPlus Hoggy("Horatio Hogg", 382288, 3000.00);

BrassPlus ddd;
ddd = Piggy; // 적당한 생성자가 있으므로 ok
RatedPlayer rp;
TableTennisPlayer ttp = rp; // 생성자 없으므로 에러

BrassPlus* eee = &Piggy; // 컴파일 에러
BrassPlus* eee = (BrassPlus*)&Piggy;
eee->ViewAcct(); // 실행 에러의 소지 다분함
```

대입, 참조, 포인터 코딩 실험

```
Brass Piggy("Porcelot Pigg", 381299, 4000.00);
BrassPlus Hoggy("Horatio Hogg", 382288, 3000.00);
Piggy.ViewAcct(); cout << endl;
Hoggy.ViewAcct(); cout << endl;

Brass aaa(Piggy);      // 대입
aaa.ViewAcct(); cout << endl;
aaa = Hoggy; //  나중에 Brass aaa1(Hoggy);로 바꿔서 실험
aaa.ViewAcct(); cout << endl; cout << endl;

Brass& bbb = Piggy; // 참조
bbb.ViewAcct(); cout << endl;
bbb = Hoggy; // 나중에 Brass& bbb1 = Hoggy; 로 바꿔서 실험 (429, 430쪽)
bbb.ViewAcct(); cout << endl; cout << endl;

Brass* ccc = &Piggy; // 포인터
ccc->ViewAcct(); cout << endl;
ccc = &Hoggy; // 나중에 Brass* ccc1 = &Hoggy;  로 바꿔서 실험
ccc->ViewAcct(); cout << endl; cout << endl;
```

함수 대입, 함수 참조, 함수 포인터

```
void func1(Brass p) { p.ViewAcct(); }
void func2(Brass& p) { p.ViewAcct();}
void func3(Brass* p) {p->ViewAcct();}
Brass ddd(Piggy); func1(ddd);
Brass ddd1(Hoggy); func1(ddd1);
Brass& eee = Piggy; func2(eee);
Brass& eee1 = Hoggy; func2(eee1);
Brass* fff = &Piggy; func3(fff);
Brass* fff1 = &Hoggy;     func3(fff1);
```

- 리스코프 대체 원리 – Liskov Substitution Principle (Barbara Liskov)
  - 원문 : 서브 타입은 언제나 기반 타입으로 교체할 수 있어야 한다.
  - C++ : 파생 클래스는 언제나 기초 클래스로 교체할 수 있어야 한다.
  - 즉, 하위 클래스는 상위 클래스와 호환성을 생각해야 한다는 것
  - 그러려면 구현은 선언을 준수하고 하위 클래스는 상위 클래스의 규약을 준수해야 함

가상 파괴자가 필요한 이유

- 831쪽, 리스팅 13.10 – delete로 파괴
- 파괴자가 가상이 아니라면 포인터형에 해당하는 파괴자가 호출되게 됨
- EC++의 규칙 7번
  - 다형성을 가진 기본 클래스에서는 소멸자를 반드시 가상 소멸자로 선언

정적 결합과 동적 결합

- 정적 결합 – 하여간 정해진 데로 하겠다!
  - 함수 이름에 따라 어떤 코드를 실행할지를 컴파일 타임에 결정
  - static binding 또는 early binding
- 동적 결합 – 봐가면서 하겠다.
  - 함수 이름에 따라 어떤 코드를 실행할지를 프로그램 실행 시간에 결정
  - dynamic binding 또는 late binding

업 캐스팅, 다운 캐스팅

- 업 캐스팅 – 파생 클래스의 참조나 포인터를 기초 클래스의 참조나 포인터로 변환하는 것
  - BrassPlus x; Brass* y = &x; Brass& z = x;
  - BrassPlusPlus xx; Brass* yy = &xx;
- 다운 캐스팅 – 기초 클래스의 참조나 포인터를 파생 클래스의 참조나 포인터로 변환하는 것
  - 명시적인 데이터 형변환 필요함
  - 기초 클래스에 없는 것을 참조하면 런타임 에러 발생 (그림 13.4)

동적 결합에 대한 논의들

- 동적 결합이 더 멋져 보인다!
  - 왜 두 종류의 결합이 있는가? (836,837쪽)
  - 왜 정적 결합이 디폴트인가?
  - 동적 결합은 어떻게 동작하는가?
- 효율성
  - 동적 결합은 실행 시간에 객체를 결정해야 하므로, 정적 결합보다 느리다.
  - 상속을 하지 않을 거라면 정적 결합으로 충분하다.
  - 사용하지 않을지 모르는 기능 때문에 미리 부담을 떠안지 않는다. – 비야네 스트롭스트룹
- 개념 모델
  - virtual 을 쓰지 않는 것으로, 파생 클래스에서 재정의되기를 원하지 않는 메쏘드들은 동적 결합을 안한다고 공공연히 알림
- 837쪽 팁 → 그렇다면 Java의 final class처럼 아예 막을 수 있는 방법은? (시험 문제)
- 그렇다면 Java의 경우, 가상함수는 어떻게 하면 되는 것일까? (역시 시험 문제)

가상 함수의 동작

- 가상 함수 테이블 (vtable) – 가상 함수의 주소 저장 (839쪽)
- 클래스에 숨겨진 멤버 – 가상 함수 테이블의 포인터
- 가상함수 실행 순서
  1. 그 객체 클래스의 vtable 주소 획득
  2. vtable에서 함수 주소 획득
  3. 함수 실행
- 자원(메모리와 속도)의 부담
  - 클래스에 대해 가상 함수 테이블
  - 객체마다 가상 함수 테이블 포인터
  - 테이블에 접근하는 단계 (indirect addressing)

가상 메쏘드(함수) 정리 (840쪽)

- virtual 로 시작 – 기초 클래스, 파생 클래스에 대해 가상 메쏘드가 됨
- 포인터나 참조를 쓰게 되면 동적 결합
- 파상 클래스에서 다시 정의하고 싶은 메쏘드는 가상 함수로 선언하자
- 생성자 – 가상 함수로 선언할 수 없다
- 파괴자 – 상속할 생각이면 가상으로 선언해야 한다. 디폴트로 만들게 하지 말고 무조건 선언하고 정의

함수 오버로딩과 파생 클래스의 오버라이딩 (842쪽)

- 어떤 함수를 파생 클래스에서 다시 정의하면 시그너처와는 무관하게 같은 이름을 가진 모든 함수를 가려버림(override)
- 따라서 상속된 메쏘드를 정의할 때는 기초 클래스의 메쏘드와 정확히 일치시켜야 혼란이 없음 → 단, 리턴 형이 기초 클래스의 참조나 포인터이면 파생 클래스의 참조나 포인터로 대체됨 (리턴 형의 공변)
- 또한 기초 클래스 선언이 오버로딩되어 있으면 파생 클래스에서는 모두 재정의하도록 한다

Protected

- 파생 클래스에서는 접근 가능
- 이건 과연 쓸만한가? → 설계자가 정한 인터페이스로만 데이터를 바꾼다는 원칙 (845쪽)
- Singleton 패턴 – 정확하게 한 개의 객체만 만들어서 호출한 쪽에게 그 객체만 리턴하게 한다

추상화 기초 클래스 (ABC)

- 구현되지 않는, 즉 선언만 하고 정의는 하지 않는, 순수 가상 함수를 가지고 있는 클래스
- 순수 가상 함수는 함수 선언 뒤에 “=0” 붙임 (849쪽) → 인터페이스의 개념
- 왜 필요한가?
  - 원과 타원의 예, 사각형과 정사각형의 예
  - BaseEllipse를 Circle과 Ellipse로 상속
- Brass와 BrassPlus 클래스에 적용 (850쪽~)

ABC 철학

- 추상화 기초 클래스만 기초 클래스가 되게 해야 한다는 주장이 있음 (856쪽 노트) – Java에서 인터페이스를 만드는 것과 비슷

파생 클래스에서 동적 메모리 할당

- 첫번째 – 파생 클래스가 new를 사용 안함 (856쪽)
  - 파생 클래스에서 명시적 파괴자, 복사 생성자, 대입 연산자 따로 구현할 필요 없음 → 기초 클래스에서 알아서 해줌
- 두번째 – 파생 클래스가 new를 사용 (858쪽)
  - 파괴자에선 자신이 new한 것만 delete
  - 복사 생성자에선 기초 클래스 복사 생성자 먼저 호출 (859쪽 중간 코드) → 인수는 파생 클래스 참조를 전달해도 무관 (업캐스팅)
  - 아무 것도 쓰지 않으면 디폴트 생성자 자동 호출됨
  - 대입연산자의 경우 자기 대입 점검 후에 바로 기초 클래스의 대입 연산자 실행 (860쪽 코드)

파생 클래스에서 프렌드 함수에 의한 연산자 오버로딩

- operator<<
  - 받은 파생 클래스 객체를 가지고, 기초 클래스로 강제로 형변환한 후 호출 (865쪽)