제15장

다형성과 가상 함수

가상 함수 기초

- Polymorphism 다형성
  - 하나의 함수에 여러 의미를 연관시킨다.
  - 가상 함수로 구현되는 객체 지향 프로그래밍의 기본 원칙

- 도형 예제
  - 여러 종류의 도형에 대한 클래스: 직사각형, 원, 타원, 등
  - 각 도형은 다른 클래스의 객체
    - 직사각형 데이터: 높이, 넓이, 중심점
    - 원 데이터: 중심점, 반지름
  - 모든 도형이 부모 클래스 Figure로부터 파생
  - 필요한 함수: draw()
    - 각 도형에 대해 다른 명령

도형 예제

- 각 클래스는 다른 draw 함수가 필요하다.
  - 각 클래스에서 "draw“가 호출될 수 있다, 그래서:

```
Rectangle r;
Circle c;
r.draw();  //Calls Rectangle class's draw
c.draw(); //Calls Circle class's draw
```

- 모든 도형에 적용되는 함수를 가지는 부모 클래스 Figure에서 도형을 화면의 가운데로 옮기는 함수 center()
  - 처음 것을 지우고 다시 그리므로, 함수 draw()를 사용
  - 복잡한 문제!
    - 어느 draw() 함수인가? 어느 클래스의 것인가?

도형 예제: 새로운 도형

- 새로운 종류의 도형: Figure 클래스에서 상속된 Triangle 클래스
  - 함수 center()는 Figure로부터 상속
  - 이것은 triangles에 대해서도 동작하나?
  - 이것은 각 도형에 대해서 다른 draw()를 사용한다!
  - 이것은 Figure::draw()를 사용한다. → triangles에 대해서 동작하지 않는다.
- 상속된 함수 center()가 함수 Figure::draw()가 아닌 함수 Triangle::draw()를 사용하기를 원한다.
  - 그러나 클래스 Triangle는 Figure::center()가 있을 때 작성되지도 않았다! "Triangle"을 모른다!

가상함수

- virtual(가상) 함수
  - “함수가 어떻게 구현되는지 모른다.”
  - “프로그램에서 사용되기 전까지 기다려라.”
  - “객체 인스턴스로부터 구현을 얻어라.”
- 사후 바인딩 또는 동적 바인딩이라고 함.
  - 가상 함수는 사후 바인딩을 구현한다.

가상 함수: 다른 예제

- 자동차 부품 상점의 판매 기록 프로그램
  - 판매 추적: 아직 모든 판매 유형을 알 수 없음
  - 처음에는 정상 가격 판매만 있고, 추후에 할인 판매, 우편 주문 등 발생
    - 가격과 세금 이외에 다른 요소에 의존적.

- 프로그램은:
  - 일일 총 판매액을 계산해야 한다.
  - 하루 중 최고 판매액과 최저 판매액을 계산해야 한다.
  - 일일 평균 판매액을 계산해야 한다.
- 모든 것은 개별 계산서로부터 계산된다.
  - 그러나 계산서를 계산하는 많은 함수는 "나중에" 추가될 것이다!
    - 다른 종류의 판매가 추가될 때!
- 그래서 "계산서를 계산"하는 함수는 가상이다

클래스 Sale 와 member function

```
class Sale
{
public:
    Sale();
    Sale(double thePrice);
    double getPrice() const;
    virtual double bill() const;
    double savings(const Sale& other) const;
private:
    double price;
};
```

```
double Sale::savings(const Sale& other) const
{
    return (bill() – other.bill());
}

bool operator < (const Sale& first, const Sale& second)
{
    return (first.bill() < second.bill());
}
```

파생 클래스 DiscountSale 정의

```
class DiscountSale   :  public Sale
{
public:
     DiscountSale();
     DiscountSale( double thePrice, double the Discount);
     double getDiscount() const;
     void setDiscount(double newDiscount);
     double bill() const;
private:
     double discount;
};
```

```
double DiscountSale::bill() const
{
     double fraction = discount/100;
     return (1 – fraction)*getPrice();
}
```

가상: 와우!

- 파생 클래스 DiscountSale가 작성되기 전에 작성된 클래스 Sale
  - 멤버 savings와 "<“는 DiscountSale 클래스를 생각하기 전에 컴파일됨
- 이제 다음과 같이 호출한다:

```
DiscountSale d1, d2;
d1.savings(d2);
```

  - savings()에서 함수 bill()을 호출→ DiscountSale 클래스의 bill()의 정의 사용

```
double Sale::savings(const Sale& other) const
{
    return (bill() – other.bill());
}

bool operator < (const Sale& first, const Sale& second)
{
    return (first.bill() < second.bill());
}
```

가상함수 사용법

- C++ 프로그램을 작성하자:
  - 가상 함수는 사후 바인딩을 구현한다.
    - 컴파일러에게 함수가 프로그램에서 사용될 때까지 “기다리라”고 말하는 것
  - 호출 객체에 기반하여 어떤 정의를 사용할지를 결정한다.
- 매우 중요한 OOP 원칙이다!

오버라이딩

- 파생 클래스에서 변경되는 가상 함수 정의
  - 이것을 “오버라이딩”되었다고 말한다.
  - 가상 함수를 변경: ***오버라이딩***
  - 가상이 아닌 함수를 변경: ***재정의***

가상 함수: 모든 것에 사용하지 않을까?

- 이제까지 보았던 것과 같이 가상 함수는 분명한 장점이 있다.
- 주요 단점: 오버헤드!
  - 많은 저장공간을 사용한다.
  - 사후 바인딩은 “그때 그때 처리”되므로, 프로그램의 실행이 느려진다.
- 그래서 가상 함수가 필요하지 않으면, 사용하지 않는다.

순수 가상 함수

- 기반 클래스는 여러 멤버 함수에 대해 “의미 있는” 정의를 가지지 못한다!
  - 이것의 목적은 단지 파생되는 것들을 위한 것이다.
- 클래스 Figure를 상기해 보자.
  - 모든 도형은 파생 클래스의 객체: 직사각형, 원, 삼각형 등
  - 클래스 Figure는 어떻게 그릴지 전혀 모른다!
- 그것을 순수 가상 함수로 만들어라:

```
virtual void draw() = 0;
```

추상 클래스

- 순수 가상 함수는 정의가 필요 없음.
  - 모든 파생 클래스가 “자신만의” 버전을 정의하도록 한다.
- 하나 이상의 가상 함수를 가진 클래스: 추상 클래스
  - 오로지 기반 클래스로 사용될 수 있고, 대상 객체를 생성할 수 없음
    - 모든 멤버의 완벽한 “정의”를 가지지 않기 때문에
- 파생 클래스가 모든 순수 가상 함수를 정의하지 않으면:
  - 파생 클래스도 추상 클래스이다.

포인터와 가상함수: 확장 형 호환성

- 주어지기를:

  Derived는 Base의 파생 클래스이다.

  – Derived 객체는 Base 형의 객체에 할당될 수 있다.

  – 그러나 반대로는 되지 않는다!

- 이전 예제를 생각해 보자:

  – DiscountSale은 Sale이다.(“is a”), 그러나 반대는 거짓이다.

확장 형 호환성 예제

- 주어진 선언:

```
Dog vdog;
Pet vpet;
```

- 멤버 변수 name과 breed가 public인 것을 주목!
  - 오로지 예제의 목적이며. 바람직하지 않음

```
class Pet
{
public:
    string name;
    virtual void print() const;
};
class Dog : public Pet
{
public:
    string breed;
    virtual void print() const;
};
```

- Dog이고("is a") pet인("is a") 어떤 것:
  - 이것들은 허용된다.

```
vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpet = vdog;
```

- 부모 형에 값을 할당할 수 있다, 그러나 반대는 안 된다.
  - pet은 dog이 아니다.("is not a")

슬라이싱 문제

- vpet에 할당된 값이 breed 필드를 잃은 것을 주목하자!
  - cout << vpet.breed; // 오류 메시지를 생성한다!
- 타당할 수 있다.
  - Dog이 Pet 변수로 이동하기 때문에 Pet 처럼 취급된다.
    - 그러므로 "dog" 성질을 가지지 않는다.
  - 흥미로운 철학적 논쟁을 일으킨다.

슬라이싱 문제 수정

- C++에서, 슬라이싱 문제는 골칫거리이다.
  - 이것은 여전히 Tiny라는 이름의 Great Dane이다.(“is a”)
  - Pet으로 다루어져도 breed를 참조하기를 원한다.
- 동적 변수에 대한 포인터로 해결할 수 있다.

슬라이싱 문제 예제

```
Pet *ppet;
Dog *pdog;
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Great Dane";
ppet = pdog;
```

- ppet가 가리키는 객체의 breed 필드에 접근할 수 없다:
  - cout << ppet->breed;//불법!
- 가상 멤버 함수를 사용해야
  - ppet->print();
  - 가상이므로, Dog 클래스의 print 멤버 함수를 호출한다!
  - C++는 호출을 “바인딩”하기 전에 객체 포인터 ppet가 실제로 어떤 것을 가리키는지를 알기 위해서 “기다린다”.

가상 소멸자

- 상기: 소멸자는 동적으로 할당된 데이터를 반환하기 위해 필요
- 다음을 생각해 보자:

```
Base *pBase = new Derived;
…
delete pBase;
```

  - Derived 클래스 객체를 가리키고 있어도 기반 클래스 소멸자를 호출한다!
  - 소멸자를 virtual로 하면 이 문제를 해결한다!
- 모든 소멸자를 가상으로 하는 것은 좋은 정책이다.

변환

- 다음을 생각해 보자:

```
Pet vpet;
Dog vdog;
…
vdog = static_cast<Dog>(vpet);  //불법!
```

- pet을 dog으로 변환할 수 없다, 그러나:

```
vpet = vdog;         // 적법!
vpet = static_cast<Pet>(vdog);  //또한 적법!
```

- 확대 변환은 된다.
  - 자손형을 조상형으로 변환

축소 변환

- 축소 변환은 위험하다!
  - 조상형을 자손형으로 변환
  - 정보가 “추가”된다고 가정하자.
  - dynamic_cast는 된다:

```
Pet *ppet;
ppet = new Dog;
Dog *pdog = dynamic_cast<Dog*>(ppet);
```

    - 적법, 그러나 위험하다!
- 축소 변환은 함정 때문에 드물게 사용된다.
  - 추가되는 모든 정보를 추적해야 한다.
  - 모든 멤버 함수가 가상이어야 한다.

가상 함수의 내부 작업

- 그것을 어떻게 사용하는지 알 필요가 없다!
  - 정보 은닉의 원칙
- 가상 함수 테이블
  - 컴파일러가 생성한다.
  - 각 가상 멤버 함수에 대한 포인터를 가진다.
  - 각 함수의 올바른 코드의 위치를 가리킨다.
- 이와 같은 클래스의 객체도 또한 포인터를 가진다.
  - 가상 함수 테이블을 가리킨다.

요약

- 사후 바인딩은 실행될 때까지 어떤 멤버 함수를 호출할지 결정을 미룬다.
  - C++에서 가상 함수가 사후 바인딩을 사용한다.
- 순수 가상 함수는 정의가 없다.
  - 최소 하나의 순수 가상 함수를 가진 클래스는 추상이다.
  - 추상 클래스에서 객체를 생성할 수 없다.
  - 다른 클래스가 파생되도록 기반 클래스에서 엄격하게 사용된다.
- 파생 클래스 객체는 기반 클래스 객체에 할당할 수 있다.
  - 기반 클래스 멤버를 읽으면; 슬라이싱 문제
  - 포인터 할당과 동적 객체로 슬라이싱 문제를 “해결”할 수 있다.
- 모든 소멸자를 가상으로 만든다.
  - 메모리가 올바르게 반환되는, 좋은 프로그래밍 습관