Chapter 15. 다형성과 가상함수

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

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

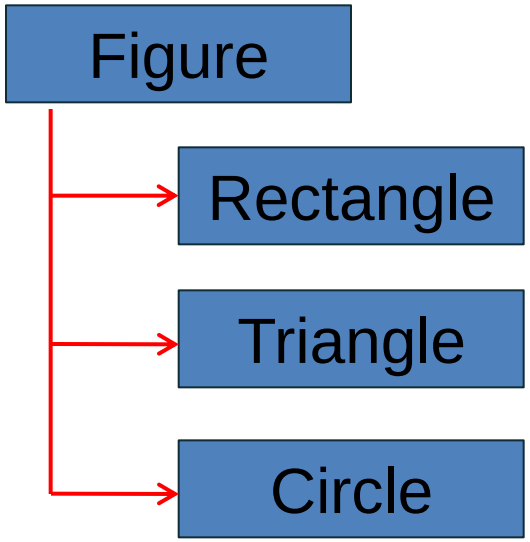
Learning Objectives

- 가상 함수(Virtual Function) 기본
  - 사후 바인딩 (late binding) /동적 바인딩(dynamic binding)
  - 가상 함수 구현
  - 가상 함수를 사용할 때
  - 추상 클래스와 순수 가상 함수
- 포인터와 가상 함수
  - 확장 형 호환
  - 축소변환과 확대변환
  - C++에서 가상 함수 구현

가상 함수(Virtual Function) 기본

- 다형성(Polymorphism)
  - 하나의 함수가 여러 의미를 가지고 사용되는 능력
  - 가상 함수는 이러한 방법을 제공
  - OOP의 기본 규칙!
- 가상(Virtual)
  - 실제 사실이 아님에도 존재하거나 본질적인 것에 영향을 미치는
- 가상 함수
  - 해당 함수가 정의되기 전에 사용됨

도형 클래스 예 (1/5)



- 여러 종류의 도형에 대한 클래스
  - rectangle, circle, ovals, triangle …
  - 각각의 도형은 다른 클래스의 객체
    - Rectangle 데이터: 높이, 폭, 중심점
    - Circle 데이터: 중심점, 반지름
- 이러한 모든 클래스는 Figure 클래스로부터 파생
- draw() 함수
  - 해당 도형마다 다른 방식이 요구됨

도형 클래스 예 (2/5)

- 각각의 클래스는 다른 draw() 함수가 필요
- 각각의 객체는 각각의 클래스의 draw() 함수 호출:

```
Rectangle r;
Circle c;
r.draw();  //Rectangle 클래스 draw() 호출
c.draw(); //Circle 클래스 draw() 호출
```

도형 클래스 예 (3/5)

- 부모 클래스 Figure는 모든 도형에 적용할 수 있는 함수를 가짐:
  center(): 도형을 화면의 중심에 옮기는 함수
  - 도형을 지우고 화면에 다시 그림
  - 따라서 Figure::center()는 다시 그리기 위해 draw() 함수를 사용
  - 복잡함!
    - 어떠한 클래스의 draw() 함수?

도형 클래스 예 (4/5)

- 새로운 도형 추가:
  class Triangle : Figure
- 함수 center()는 Figure에서 상속
  - 이 함수가 Triangle에 적용되는가?
  - center()는 draw()를 사용하고 draw()는 도형마다 내용이 다름!
  - Figure::draw() → Triangle에 적절하게 동작하지 않음
- 상속받은 함수 center()에서 Figure::draw()가 아니고 Triangle::draw() 함수로 동작 하는 것이 요구됨
  - Figure::center()가 작성될 때 Triangle 클래스에 대한 고려를 하지 않음!

도형 클래스 예 (5/5)

- 가상 함수로 해결
- 컴파일러에게:
  - 함수가 어떻게 구현되는지 모름
  - 프로그램이 사용될 때까지 대기
  - 객체의 인스턴스로부터 구현을 얻어라!
- 사후 바인딩 또는 동적 바인딩
  - 가상 함수의 구현은 동적 바인딩 됨

가상 함수의 다른 예 (1/2)

- 자동차 부품 상점의 판매 기록 관리 프로그램
  - 판매 정보의 지속적 관리가 요구됨
  - 판매 유형의 예측이 어려움
  - 우선, 정상적인 가격에 부품을 파는 소매 판매 고려
  - 나중에: 할인 판매, 우편판매 등이 고려됨

가상 함수의 다른 예 (2/2)

- 프로그램 요구사항:
  - 일일 총 판매액 계산
  - 하루 중 최고/최처 판매액 계산
  - 하루의 평균 판매액 계산
- 각각의 판매는 각각의 청구서로 계산
  - 청구서를 계산하는 함수는 판매 유형이 결정될 때까지 첨부되지 않음
- 다양한 상황의 수용을 위해 청구서를 계산하는 함수를 가상 함수로 함!

Sale 클래스 정의

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

멤버 함수 savings()와 연산자 <

- double Sale::savings(const Sale& other) const
  {
  return (bill() – other.bill());
  }
- bool operator < (const Sale& first,
  const Sale& second)
  {
  return (first.bill() < second.bill());
  }
- 둘 모두 멤버함수 bill()을 사용함에 유의!!

Sale 클래스

- Sale 클래스는 추가된 할인이나 경비가 없는 단일 품목에 대한 단순 판매를 정의
- 멤버 함수 bill()의 선언에 "virtual" 예약어 추가
  - 영향: 사후, Sale 클래스의 파생 클래스는 자신에 적합한 함수 버전의 정의가 가능
  - Sale 클래스의 다른 멤버 함수는 파생 클래스 객체 버전의 함수를 사용!
  - 자동적으로 Sale 클래스 버전의 함수를 사용하지 않음!

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;
  };

DiscountSale 클래스의 bill() 함수 구현 (1/2)

- double DiscountSale::bill() const
  {
  double fraction = discount/100;
  return (1 – fraction)*getPrice();
  }
- 함수 정의에서는 "virtual“이 명시되지 않음
  - 파생 클래스에서는 자동적으로 가상함수가 됨

DiscountSale 클래스의 bill() 함수 구현 (2/2)

- 기반 클래스의 가상 함수:
  - 파생 클래스에서 자동으로 가상 함수가 됨
- 파생 클래스 선언(인터페이스)
  - “virtual”을 명시하지 않아도 됨
  - 일반적으로 가독성을 위해 추가

파생 클래스 DiscountSale

- DiscountSale 클래스의 멤버 함수 bill()은 Sale 클래스의 구현과 다르게 정의됨
- 멤버 함수 *savings()와 연산자 <*
  - DiscountSale 클래스 객체에 대하여 Sale 클래스에 있는 버전대신 DiscountSale 클래스의 bill() 함수 사용!

가상 함수 효과

- Sale 클래스는 파생 클래스 DiscountSale 이전에 작성됨
  - 멤버 함수 savings()와 < 연산자는 DiscountSale 클래스가 고려되기 전에 컴파일 됨
- 다음의 호출고려:
  DiscountSale d1, d2;
  d1.savings(d2);
  - savings() 함수내의 bill()의 호출에서 DiscountSale 클래스의 정의 버전을 사용해야 하는 것을 알고 있음
- Powerful!

가상 함수의 동작 방법

- 사후 바인딩과 연관됨
  - 가상 함수는 사후 바인딩으로 구현
  - 컴파일러는 함수가 사용될 때까지 대기
  - 호출하는 객체에 대응하여 구현부분을 결정함
- 매우 중요한 OOP 규칙!

오버라이딩 (Overriding)

- 파생 클래스에서 가상 함수 정의가 변경된 경우
  - 오버라이딩
- 재정의(redefine)과 유사 → 혼용됨
  - 일반 함수
- 정확한 구분(상속에서):
  - 가상 함수의 변경: *오버라이딩*
  - 일반 함수 변경: *재정의*

가상함수의 장/단점

- 장점 → 앞의 예
- 단점: overhead!
  - 많은 저장 공간을 사용
  - 사후 바인딩이 실행될 때 까지 프로그램은 대기
- 가상함수의 사용이 절대적으로 옳은 것은 아님 → 가상 함수의 장점이 없을 경우 → 일반 함수로 구현

순수 가상 함수

- 기반 클래스의 함수의 구현이 의미가 없을 경우!
  - 파생 클래스에서 항상 오버라이딩 되어야 할 경우
- Figure 클래스
  - 모든 도형은 파생 클래스의 객체
    - Rectangles, circles, triangles, …
  - Figure 클래스의 draw() 정의는 너무 추상적!
- 순수 가상 함수(pure virtual function)로 만듦:
  virtual void draw() = 0;

추상 기반 클래스 (Abstract Base Class)

- 순수 가상 함수는 정의가 필요 없음
  - 모든 파생 클래스는 각각의 정의를 가짐
- 하나 이상의 순수 가상 함수를 가진 클래스: 추상 기반 클래스
  - 기반 클래스로만 사용
  - 객체를 생성하지 않음
    - 따라서 모든 멤버 함수의 완벽한 정의가 필요하지 않음!
- 추상 기반 클래스로부터 파생된 클래스에서 순수 가상 함수를 정의하지 않으면 → 이 클래스도 추상 기반 클래스

확장 형 호환

- “class Derived : Base” 이면
  - Derived 클래스 객체는 Base 클래스 객체에 할당 가능
  - 역은 성립하지 않음!
- 예:
  - DiscountSale 은 Sale 이지만, 역은 성립하지 않는다

확장 형 호환의 예

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

Pet 클래스와 Dog 클래스

- 다음의 선언을 고려:
  Dog vdog;
  Pet vpet;
- 멤버 변수 name과 breed는 public!
  - 설명을 위한 예제!

Pet 클래스와 Dog 클래스 사용

- dog "is a" pet:
  - vdog.name = "Tiny";
    vdog.breed = "Great Dane";
    vpet = vdog;
  - 위의 예는 허용됨
- 부모 형에 할당 가능
  - 역은 성립되지 않음
  - A pet "is not a" dog

슬라이스 문제 (Slicing Problem)

- vpet에 할당되면서 breed가 손실됨!
  - cout << vpet.breed;
    - ERROR msg!
  - 슬라이스 문제
- Dog 형이 Pet 형으로 변환
  - Dog 형 고유의 속성을 없어짐

슬라이스 문제 해결

- C++에서 , 슬라이스는 성가신 문제
- Pet 형으로 변환 되었지만 breed를 사용하고 싶을 경우
- 포인터와 동적 변수 사용

슬라이스 문제 예 (1/2)

- Pet *ppet;
  Dog *pdog;
  pdog = new Dog;
  pdog->name = "Tiny";
  pdog->breed = "Great Dane";
  ppet = pdog;
- ppet으로 breed에 접근이 불가능:
  cout << ppet->breed; //ILLEGAL!

슬라이스 문제 예 (2/2)

- 가상 멤버 함수 사용:
  ppet->print();
  - Dog 클래스의 print() 멤버 함수 호출!
    - 가상 함수
  - C++은 ppet이 어떠한 객체를 포인팅 하는지 기다리고 있다가 해당 객체의 함수를 동적 바인딩 함

가상 소멸자

- 소멸자는 동적으로 할당된 메모리의 해제가 필요함
- 다음을 고려:

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

  - 파생 클래스의 객체를 포인팅하고 있다고 하더라도 기반 클래스의 소멸자가 호출!
  - 소멸자를 ***"virtual" 로***! → 파생 클래스의 소멸자가 호출됨
- 모든 소멸자를 가상 소멸자로 하는 것은 좋은 정책!

변환

- 다음을 고려:

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

- 다음은 허용됨:

```
vpet = vdog;            // Legal!
vpet = static_cast<Pet>(vdog);  //Also legal!
```

- 확대변환(upcasting) is OK
  - 자손 형을 조상형으로

축소변환

- 축소변환(Downcasting)은 위험!
  - 조상형을 자손형으로 : 가정된 정보를 추가
  - dynamic_cast:
    Pet *ppet;
    ppet = new Dog;
    Dog *pdog = dynamic_cast<Dog*>(ppet);
    - 허용, 위험함!
- 축소변환은 잘 사용하지 않음
  - 추가되는 정보를 주시
    - 실패 시 NULL 리턴
  - 모든 기반 클래스의 함수는 virtual 이어야 함

가상 함수의 내부 동작

- 가상 함수 테이블
  - 컴파일러가 생성
  - 각각의 가상 함수에 대해 포인터를 가지고 있음
  - 포인터는 그 멤버 함수에 대한 올바른 코드에 위치
    - 가상 함수가 상속되고 구현되지 않으면 → 부모클래스 멤버함수를 포인팅
    - 새로운 정의가 생성 → 해당 형 정의를 포인팅
  - 새로운 객체가 생성되면 → 가상 함수 테이블을 포인팅
- 모든 것이 자동으로 동작
  - OOP 정보 은닉 규칙
  - 우리는 사용할 뿐~!!

요약 1

- 동적(사후) 바인딩은 멤버 함수의 어떤 버전이 적당한지에 대한 것을 런타임에 결정
  - C++에서 가상 함수가 동적 바인딩을 사용
- 순수 가상 함수는 정의가 없음
  - 하나 이상의 순수 가상 함수를 가진 클래스를 추상 클래스라 함
  - 추상 클래스 자체로 객체를 생성하지 않음
  - 파생 클래스의 기반 클래스로만 사용됨

요약 2

- 파생 클래스 객체는 기반 클래스에 할당이 가능
  - 기반 클래스에 있지 않은 멤버는 손실 → 슬라이스 문제
- 포인터 할당과 동적 객체
  - 슬라이스 문제 해결
- 소멸자를 가상으로 하는 것
  - 좋은 프로그래밍 습관
  - 메모리 회수를 명확하게 함

Q&A