객체지향설계
Fall 2018

임성수 교수

Week 10: Polymorphism & Virtual Functions

수업 내용

1. 상속과 포인터/참조자
2. 동적 바인딩
3. VTABLE
4. 순수 가상 함수
5. 가상 소멸자

공지

평가 기준

- 시험: 중간고사 (25%), 기말고사 (25%)
- 프로젝트: 제안서(10%), 중간발표(10%), 최종발표/보고서(10%)
- 출석 (10%), 실습 (10%)

중간고사 결과

01분반:

평균: 70.69 / 표준편차: 13.31

최고점: 94점 (2명)

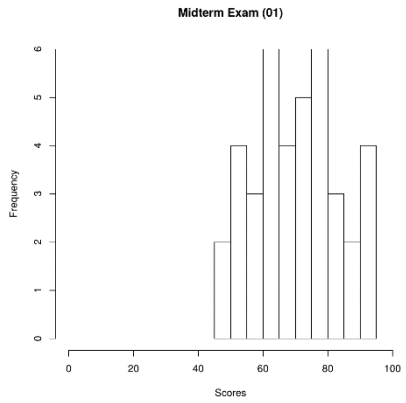
상위 15%: 87점

상위 30%: 79점

상위 50%: 72점

상위 70%: 63점

상위 85%: 56점



02분반:

평균: 65.77 / 표준편차: 14.15

최고점: 94점 (1명)

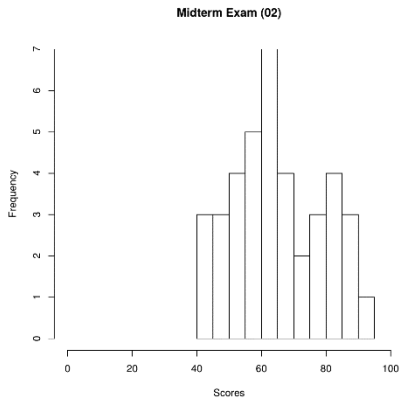
상위 15%: 82점

상위 30%: 73점

상위 50%: 64점

상위 70%: 59점

상위 85%: 52점



공지

향후 진행 계획

10주차 - 11/09 금	다형성과 가상 함수	중간고사 채점 확인
11주차 - 11/12 월	실습	
11주차 - 11/16 금	템플릿과 예외 처리	(창의SW축전으로 인한 조정 필요)
12주차 - 11/19 월	프로젝트 중간 발표	
12주차 - 11/23 금	~~휴강~~	(출장으로 인한 휴강)
13주차 - 11/26 월	실습	
13주차 - 11/30 금	디자인 패턴 소개	
14주차 - 12/03 월	실습	
14주차 - 12/07 금	주요 디자인 패턴	
추석보충 - 12/10 월	기말고사	
15주차 - 12/17 월	프로젝트 최종 발표	최종 보고서 제출
15주차 - 12/21 금	종강	최종 성적 확인

1. 상속과 포인터/참조자

객체 포인터/참조자

- 객체의 주소값을 저장할 수 있는 포인터
- **상속과 포인터**: 기존 클래스의 포인터는 클래스 객체의 주소뿐만 아니라 이를 상속하는 파생 클래스 객체의 주소값도 저장 가능
- **상속과 참조자**: 기존 클래스의 참조자는 클래스 객체뿐만 아니라 이를 상속하는 파생 클래스의 객체도 참조 가능

1. 상속과 포인터/참조자

[예제] 객체 포인터

```
#include "pch.h";
#include <iostream>
using namespace std;

class Person {
public:
  void Sleep() {
    cout << "Sleep" << endl;
  }
};
class Student :public Person {
public:
  void Study() {
    cout << "Study" << endl;
  }
};
class PartTimeStd :public Student {
public:
  void Work() {
    cout << "Work" << endl;
  }
};
```

```
int main() {
  Person* p1 = new Person;
  Person* p2 = new Student;
  Person* p3 = new PartTimeStd;

  p1->Sleep();
  p2->Sleep();
  p3->Sleep();

  return 0;
}
```

실행 결과

```
Sleep
Sleep
Sleep
```

1. 상속과 포인터/참조자

[예제] 객체 참조자

```
#include "pch.h";
#include <iostream>
using namespace std;

class Person {
public:
  void Sleep() {
    cout << "Sleep" << endl;
  }
};
class Student :public Person {
public:
  void Study() {
    cout << "Study" << endl;
  }
};
class PartTimeStd :public Student {
public:
  void Work() {
    cout << "Work" << endl;
  }
};
```

```
int main() {
  PartTimeStd p;
  Student& ref1 = p;
  Person& ref2 = p;

  p.Sleep();
  ref1.Sleep();
  ref2.Sleep();

  return 0;
}
```

실행 결과

```
Sleep
Sleep
Sleep
```

1. 상속과 포인터/참조자

객체 포인터/참조자의 권한

- 포인터를 통해서 접근할 수 있는 객체 멤버의 영역
- **상속과 포인터**: 파생 클래스의 객체 포인터는 파생 클래스의 멤버와 파생 클래스가 상속받은 기존 클래스의 멤버만 접근 가능
- **상속과 참조자**: 파생 클래스의 참조자는 파생 클래스의 멤버와 파생 클래스가 상속받은 기존 클래스의 멤버만 접근 가능

```
int main() {
  Person* p3 = new PartTimeStd;

  p3->Sleep();
  p3->Study();
  p3->Work();

  return 0;
}
```

```
int main() {
  Person& ref2 = p;

  ref2.Sleep();
  ref2.Study();
  ref2.Work();

  return 0;
}
```

class "Person"에 "Study"가 없습니다.
class "Person"에 "Work"가 없습니다.

1. 상속과 포인터/참조자

업캐스팅 (Upcasting)

- 파생 클래스에서 기존 클래스로의 형 변환

다운캐스팅 (Downcasting)

- 기존 클래스의 포인터/참조자를 파생클래스의 포인터/참조자로 형 변환
- **강제 형 변환**이기 때문에 실행 에러가 발생할 수 있음

```
int main() {
  Person* p3 = new PartTimeStd;
  p3->Sleep();

  Student *p4 = (Student*)p3;
  p4->Study();

  return 0;
}
```

업캐스팅

다운캐스팅
(Student*)를 생략할 경우: "Person" 형식의 값을 사용하여 "Student *" 형식의 엔터티를 초기화할 수 없습니다.

2. 동적 바인딩

오버라이딩 (Overriding)

- 기존 클래스에 선언된 멤버와 같은 형태의 멤버를 파생 클래스에서 선언
- **재정의**: 기존 클래스의 멤버를 가리는 효과
- 보는 시야(Pointer)에 따라서 접근하는 멤버가 달라짐

```
#include "pch.h";
#include <iostream>
using namespace std;

class Base {
public:
  void ftn() {
    cout << "Base" << endl;
  }
};
```

```
class Derived :public Base {
public:
  void ftn() {
    cout << "Derived" << endl;
  }
};
int main() {
  Derived d;
  d.ftn();

  return 0;
}
```

실행 결과

```
Derived
```

2. 동적 바인딩

[예제] 포인터와 오버라이딩

```
#include "pch.h";
#include <iostream>
using namespace std;

class Base {
public:
  void ftn() {
    cout << "Base" << endl;
  }
};
class Derived :public Base {
public:
  void ftn() {
    cout << "Derived" << endl;
  }
};
```

```
int main() {
  Derived* d = new Derived;
  d->ftn();

  Base* b = d;
  b->ftn();

  return 0;
}
```

실행 결과

```
Derived
Base
```

Lecture 3 예제:

```
int main(){
  Animal* cat = new Cat();
  cat->Crying();
};
```

```
엉엉
```

2. 동적 바인딩

다형성 (Polymorphism)

- 하나의 함수가 여러 의미를 가지고 사용되는 능력
- 가상 함수는 이러한 방법을 제공
- 객체지향 프로그래밍의 기본 규칙

가상 함수 (Virtual function)

- 기존 클래스에서 **virtual** 키워드를 붙여서 정의하면 가상 함수가 됨
- 파생 클래스에서 함수를 오버라이딩(재정의)하여 활용 가능

2. 동적 바인딩

[예제] 가상 함수 사용

```
#include "pch.h";
#include <iostream>
using namespace std;

class Base {
public:
  virtual void ftn() {
    cout << "Base" << endl;
  }
};
class Derived :public Base {
public:
  void ftn() {
    cout << "Derived" << endl;
  }
};
```

```
int main() {
  Derived* d = new Derived;
  d->ftn();

  Base* b = d;
  b->ftn();

  return 0;
}
```

실행 결과

```
Derived
Derived
```

Lecture 3 예제:

```
int main(){
  Animal* cat = new Cat();
  cat->Crying();
};
```

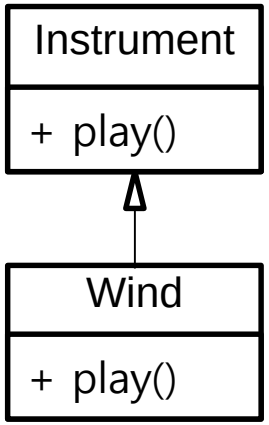
```
야옹
```

2. 동적 바인딩

바인딩(Binding)

- 함수 호출을 함수의 몸체와 연결하는 것
- **정적 바인딩**(early-/static- binding)
  컴파일(compile)시 호출되는 함수 결정, 실행 속도 빠름
- **동적 바인딩**(late-/dynamic- binding)
  실행(runtime)시 호출되는 함수 결정, 가상 함수 사용으로 융통성이 큼

2. 동적 바인딩



[예제] 업캐스팅

- 파생 클래스로부터 기존 클래스로의 형 변환

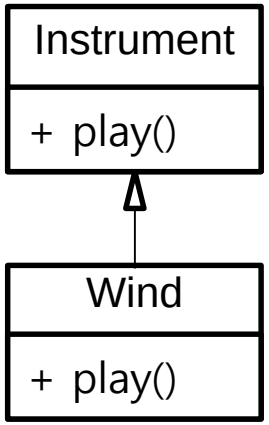
```
// Instrument.cpp
#include "pch.h"
#include <iostream>
using namespace std;

enum note {middleC, Csharp, Cflat};

// Wind (관악기) is a Instrument (악기)
// because they have the same interface:
class Instrument {
public:
  // NOT virtual function:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
```

```
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i){
  i.play(middleC);
}

int main(){
  Wind flute;
  tune(flute); // Upcasting
}
```

실행 결과

```
Instrument::play
```

2. 동적 바인딩



[예제] 동적 바인딩

- 호출 시 오버라이딩된 함수가 있는지 확인 후 실행

```
// Instrument.cpp
#include "pch.h"
#include <iostream>
using namespace std;

enum note {middleC, Csharp, Cflat};

// Wind (관악기) is a Instrument (악기)
// because they have the same interface:
class Instrument {
public:
  // Virtual function:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
```

```
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i){
  i.play(middleC);
}

int main(){
  Wind flute;
  tune(flute); // Upcasting
}
```

실행 결과

```
Wind::play
```

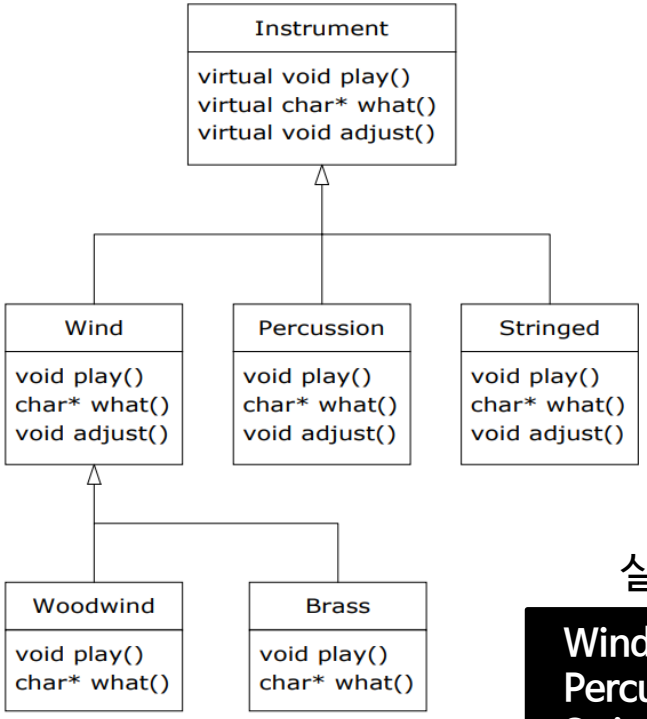
2. 동적 바인딩

[예제] 동적 바인딩

- Instrument에서 play는 가상 함수
- 각 파생 클래스마다 play를 재정의

```
void tune(Instrument& i){
  i.play(middleC);
}
```

```
int main(){
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
}
```



실행 결과

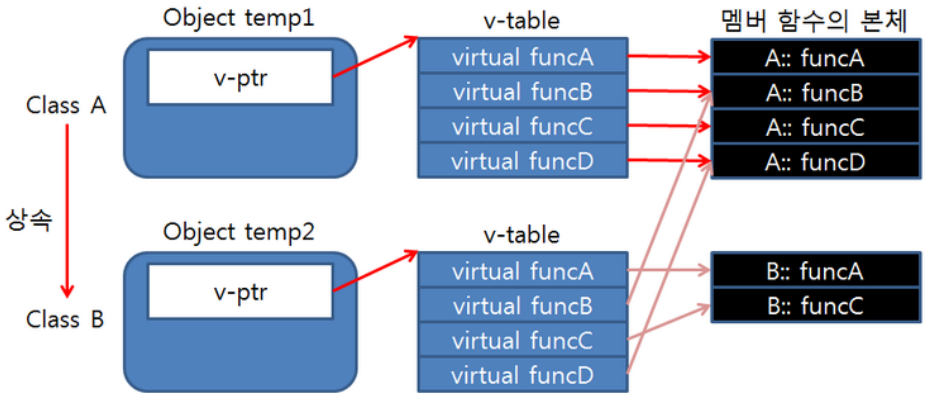
```
Wind::play
Percussion::play
Stringed::play
Brass::play
Woodwind::play
```

3. VTABLE

가상 함수 동작원리

- **가상 함수 테이블(VTABLE)**
  - **가상 함수의 주소를 모아**둔 함수에 대한 포인터 배열
  - 가상 함수를 갖는 **클래스마다 가상 함수 테이블 생성**
- **가상 함수 테이블 포인터 (VPTR)**
  - VTABLE의 주소를 저장하는 포인터
  - 가상 함수를 갖는 클래스의 객체 마다 생성
- **상속에서의 가상 함수 테이블**
  - 파생 클래스는 기존 클래스의 가상 함수 테이블을 상속 받아서 수정, 확장
  - 오버라이딩한 함수의 주소로 수정

3. VTABLE



[예제] VTABLE

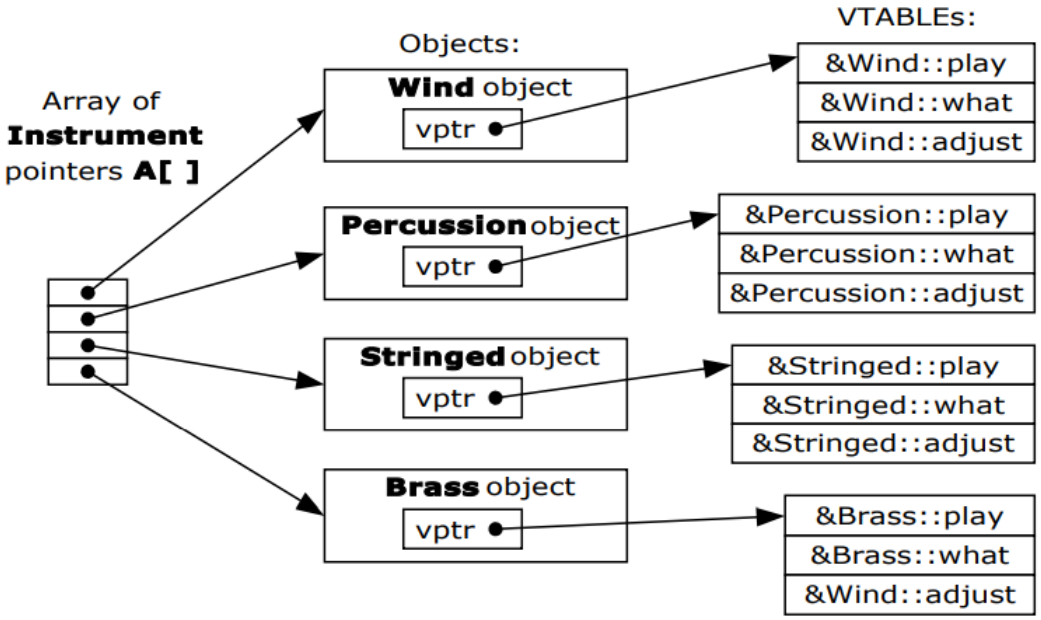
```
#include "pch.h";
#include <iostream>
using namespace std;

class A {
public:
  virtual void funcA(void) { cout << "A::funcA" << endl; }
  virtual void funcB(void) { cout << "A::funcB" << endl; }
  virtual void funcC(void) { cout << "A::funcC" << endl; }
  virtual void funcD(void) { cout << "A::funcD" << endl; }
};

class B :public A {
public:
  virtual void funcA(void) { cout << "B::funcA" << endl; }
  virtual void funcC(void) { cout << "B::funcC" << endl; }
};

int main() {
  B b;
  return 0;
}
```

3. VTABLE

[예제] VTABLE

- 포인터 배열 A에 대한 가상 함수 테이블

```
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};
```



4. 순수 가상 함수

순수 가상 함수(Pure-)

- 기능을 구현하지 않은 가상 함수
- **오버라이딩을 강제하는 효과**: 재정의해야만 의미를 가짐
- **추상 클래스**: 가상화된 멤버 함수를 가진 클래스
  - 일반(concrete) 클래스와 달리 객체를 인스턴스로 갖지 못함
  - 파생 클래스를 통해 오버라이딩해야 일반 클래스가 될 수 있음
  - 추상적인 형태만 제안하고, 실제 구현은 파생 클래스에서 이뤄짐

```
class Instrument {
public:
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  virtual void adjust(int) = 0;
};
```

재정의하지 않고 Instrument 클래스 객체 생성하려고 할 때:
컴파일 에러 - 추상 클래스를 인스턴스화할 수 없습니다.

5. 가상 소멸자

객체 포인터의 소멸

- 객체 소멸 시 파생 클래스의 소멸자를 자동으로 호출하지 않음

```
#include "pch.h";
#include <iostream>
using namespace std;

class Base1 {
public:
  ~Base1() { cout << "~Base1()" << endl; }
};
class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()" << endl; }
};
```

```
class Base2 {
public:
  ~Base2() { cout << "~Base2()" << endl; }
};
class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()" << endl; }
};

int main() {
  Base1* bp = new Derived1;
  delete bp;
  Base2* b2p = new Derived2;
  delete b2p;
}
```

실행 결과

```
~Base1()
~Base2()
```

5. 가상 소멸자

가상 소멸자(Virtual-)

- virtual 키워드를 써주면 객체 소멸 시 파생 클래스의 소멸자도 함께 호출

```
#include "pch.h";
#include <iostream>
using namespace std;

class Base1 {
public:
  ~Base1() { cout << "~Base1()" << endl; }
};
class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()" << endl; }
};
```

```
class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()" << endl; }
};
class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()" << endl; }
};

int main() {
  Base1* bp = new Derived1;
  delete bp;
  Base2* b2p = new Derived2;
  delete b2p;
}
```

실행 결과

```
~Base1()
~Derived2()
~Base2()
```

질문 및 답변