C++ 프로그래밍

김 형 기

hk.kim@jbnu.ac.kr

Today

- 다형성

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

다형성

Polymorphism

- 다형성과 동적 바인딩
- 가상 함수
- 기본 클래스의 포인터 / 참조자
- override/final 지정자
- 순수 가상 함수와 추상 클래스
- 추상 클래스와 인터페이스

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Polymorphism

- 다형성과 동적 바인딩
- 가상 함수
- 기본 클래스의 포인터/참조자
- override/final 지정자
- 순수 가상 함수와 추상 클래스
- 추상 클래스와 인터페이스

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Polymorphism

중요!

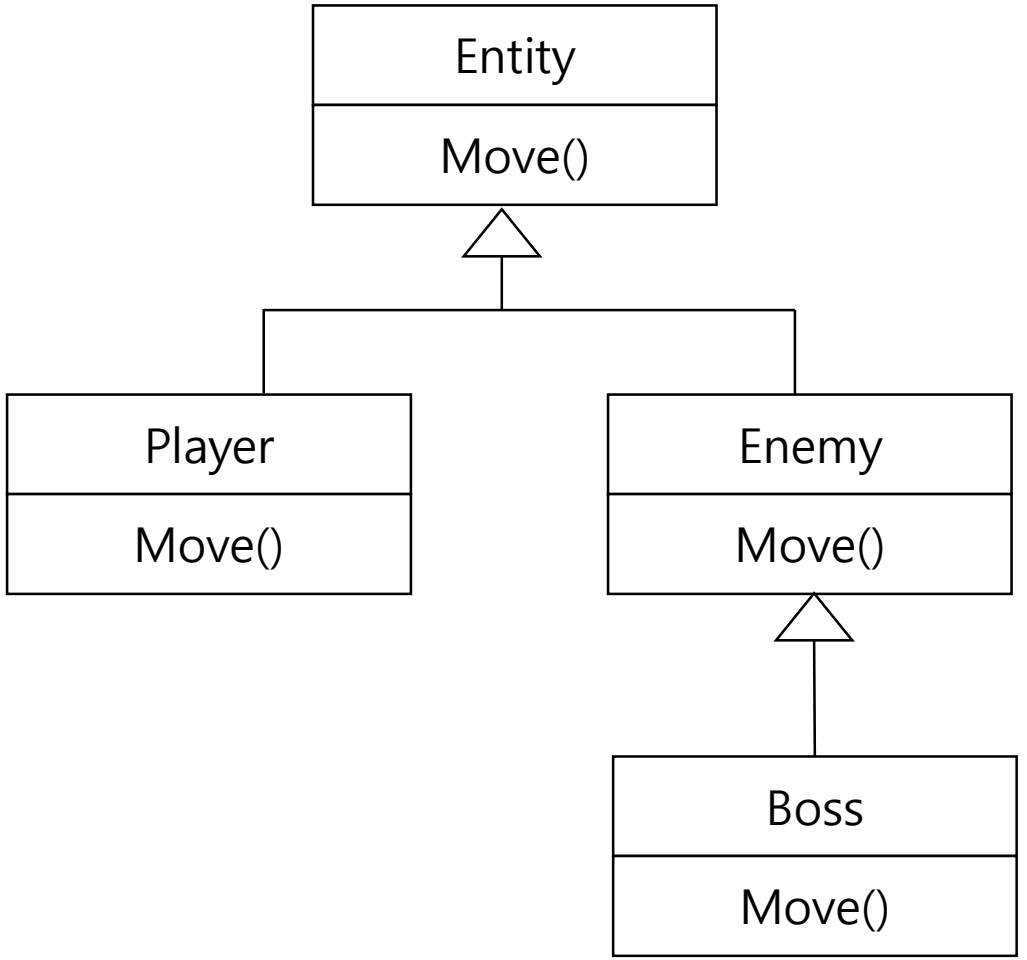
- 다형성
  - 정적 바인딩 (Compile-time) (함수 오버로딩, 연산자 오버로딩)
  - 동적 바인딩 (Run-time)
- 런타임 다형성
  - 런타임에서 같은 함수에 대해 다른 의미를 부여 = 함수의 오버라이딩
- 추상화된 프로그래밍을 가능하게 함
- C++에서 런타임 다형성의 구현을 위해 아래와 같은 조건이 필요
  - 상속
  - 기본 클래스 포인터 또는 참조자
  - 가상 함수

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Polymorphism

- 정적 바인딩의 예시 1

```
1  Entity entity{0,0};
2  entity.Move(1,1); //Entity::Move()
3
4  Player player{0,0,2};
5  player.Move(1,1); //Player::Move()
6
7  Enemy enemy{0,0,2};
8  enemy.Move(1,1); //Enemy::Move()
9
10 Boss boss{0,0,2};
11 boss.Move(1,1); //Boss::Move()
12
13 Entity *ePtr = new Boss{0,0,2};
14 ePtr→Move(1,1); //Entity::Move() !!!
```



*is-A 관계(타입을 두 개 갖게 됨)를 생각해보면, 가능한 명령문.
기본 클래스를 사용하는 곳에는 항상 유도 클래스도 사용 가능함

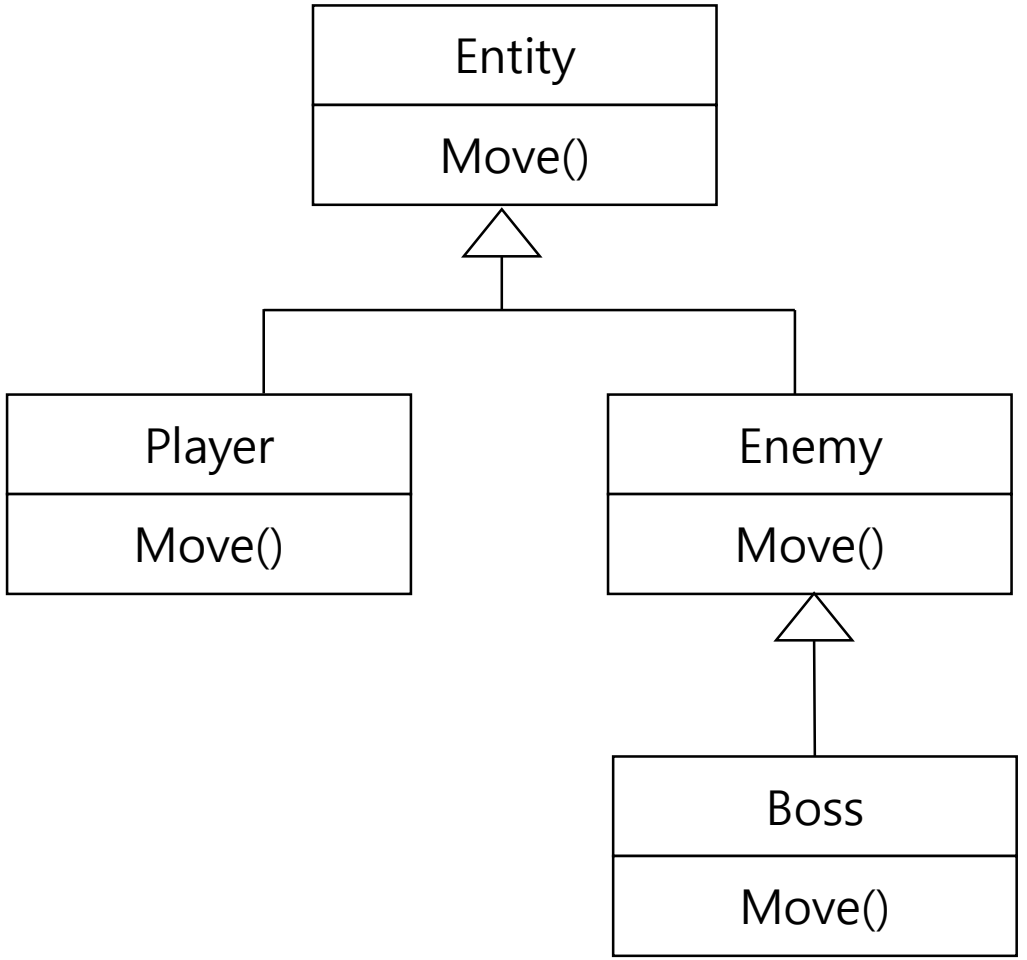
문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Polymorphism

중요!

- 정적 바인딩의 예시 1

```
1  Entity entity{0,0};
2  entity.Move(1,1); //Entity::Move()
3
4  Player player{0,0,2};
5  player.Move(1,1); //Player::Move()
6
7  Enemy enemy{0,0,2};
8  enemy.Move(1,1); //Enemy::Move()
9
10 Boss boss{0,0,2};
11 boss.Move(1,1); //Boss::Move()
12
13 Entity *ePtr = new Boss{0,0,2};
14 ePtr→Move(1,1); //Entity::Move() !!!
```



**컴파일러는 ePtr의 선언 타입(Entity*)을 기준으로 호출할 함수를 미리 결정함!*
== 정적 바인딩

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Polymorphism

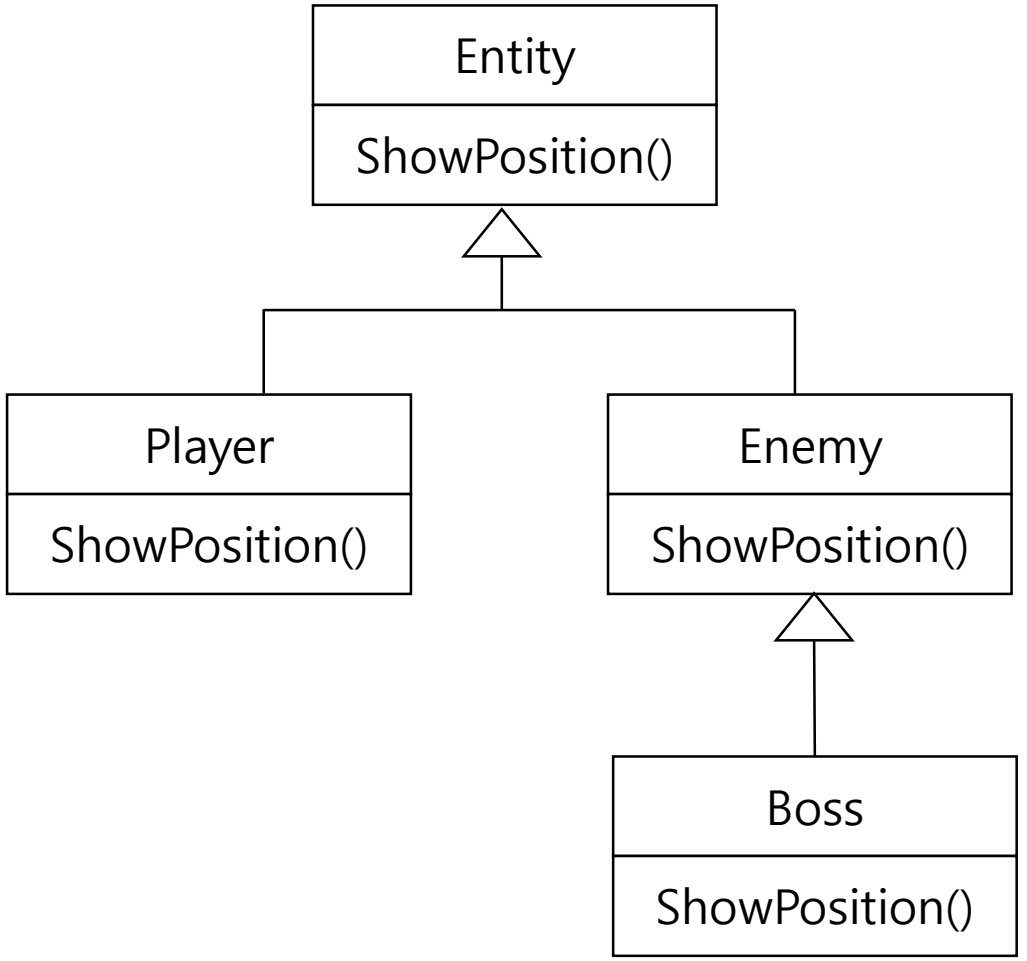
- 정적 바인딩의 예시 2

```cpp
void DisplayPosition(const Entity& e)
{
    e.ShowPosition();   //Entity::ShowPosition() called
}

Entity entity{0,0};
DisplayPosition(entity);

Player player{0,0,2};
DisplayPosition(player);

Enemy enemy{0,0,2};
DisplayPosition(enemy);
```



Polymorphism

중요!

- 정적 바인딩의 예시 2

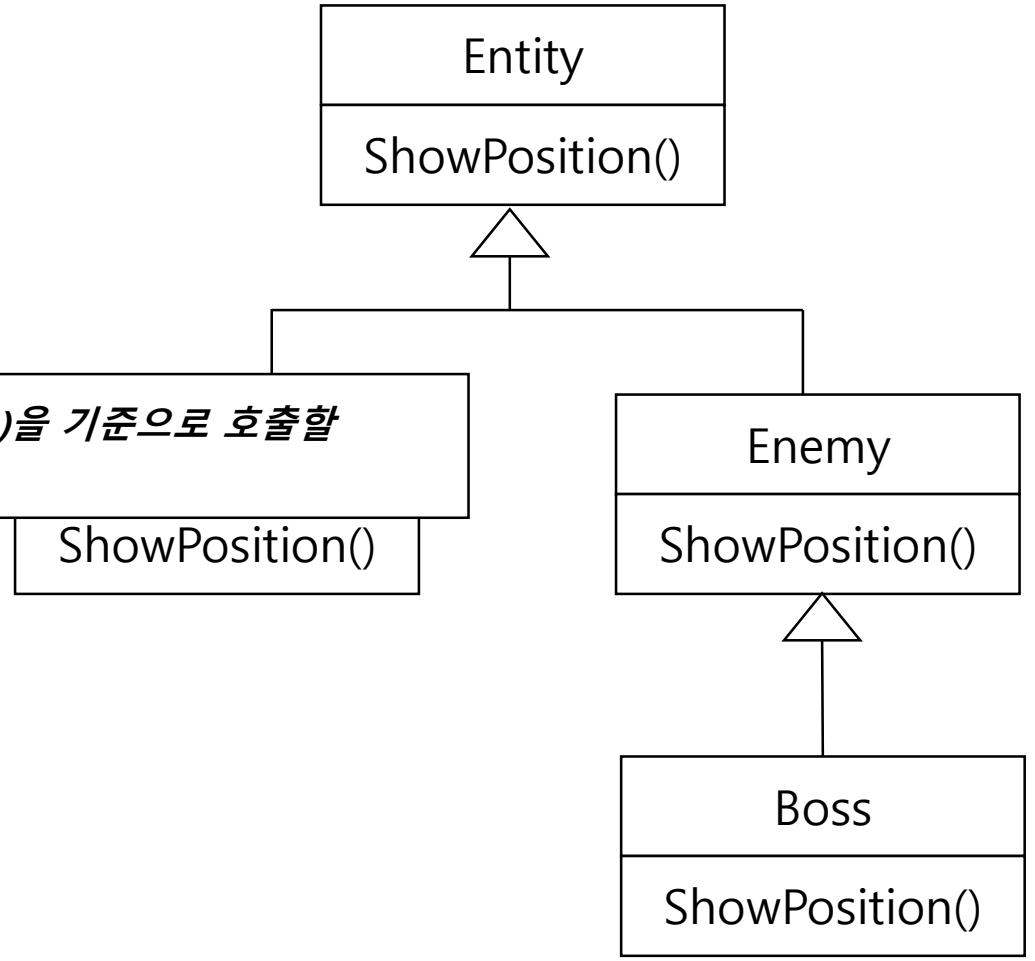
```cpp
void DisplayPosition(const Entity& e)
{
    e.ShowPosition();    //Entity::ShowPosition() called
}

Entity entity{0,0};
DisplayPosition(entity);

Player player{0,0,2};
DisplayPosition(player);

Enemy enemy{0,0,2};
DisplayPosition(enemy);
```

**컴파일러는 e의 선언 타입(Entity&)을 기준으로 호출할 함수를 미리 결정함!*



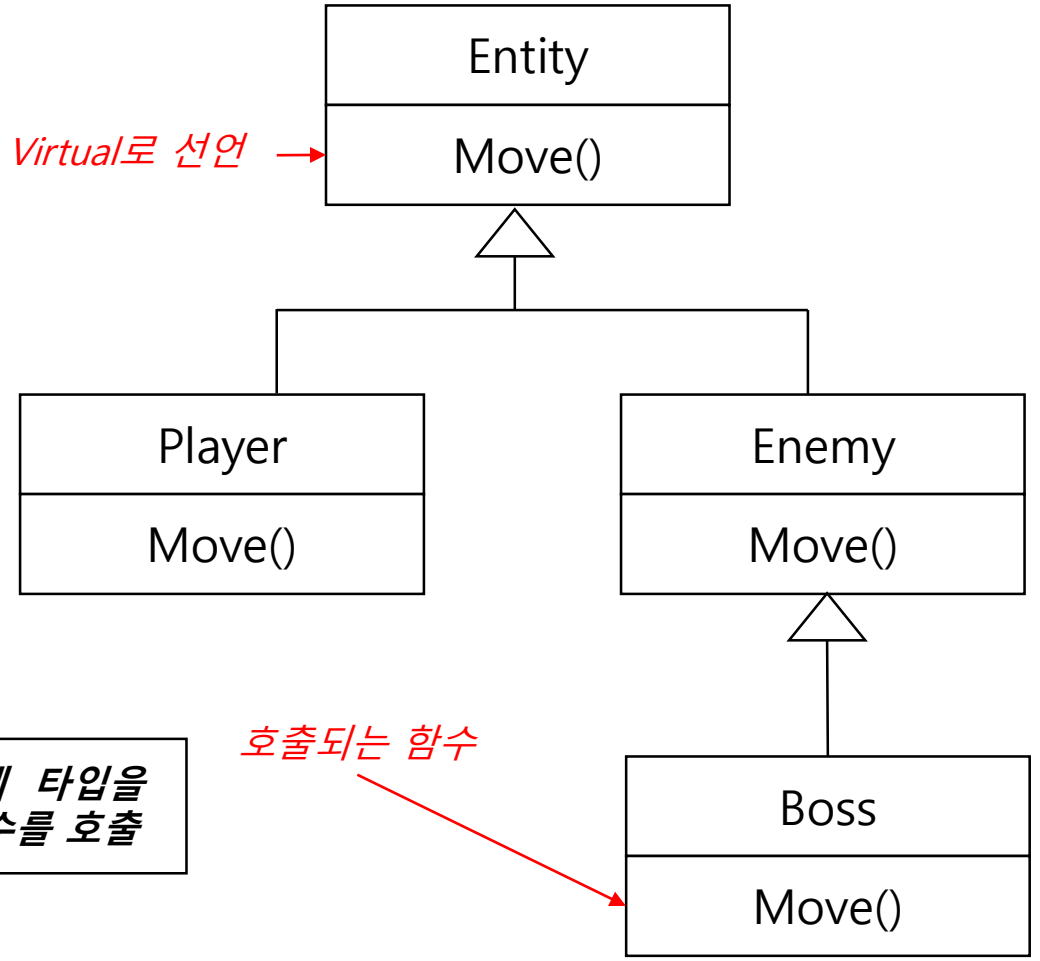
문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Polymorphism

- 동적 바인딩의 예시 1
  - (동적 바인딩을 구현하는 법은 뒤쪽에서 설명)

```
1  Entity entity{0,0};
2  entity.Move(1,1); //Entity::Move()
3
4  Player player{0,0,2};
5  player.Move(1,1); //Player::Move()
6
7  Enemy enemy{0,0,2};
8  enemy.Move(1,1); //Enemy::Move()
9
10 Boss boss{0,0,2};
11 boss.Move(1,1); //Bos
12
13 Entity* ePtr = new Boss{0,0,2};
14 ePtr→Move(1,1); //Boss::Move() !!!
```

컴파일러는 런타임에 ePtr의 실제 타입을 확인하고, 해당 클래스의 멤버 함수를 호출

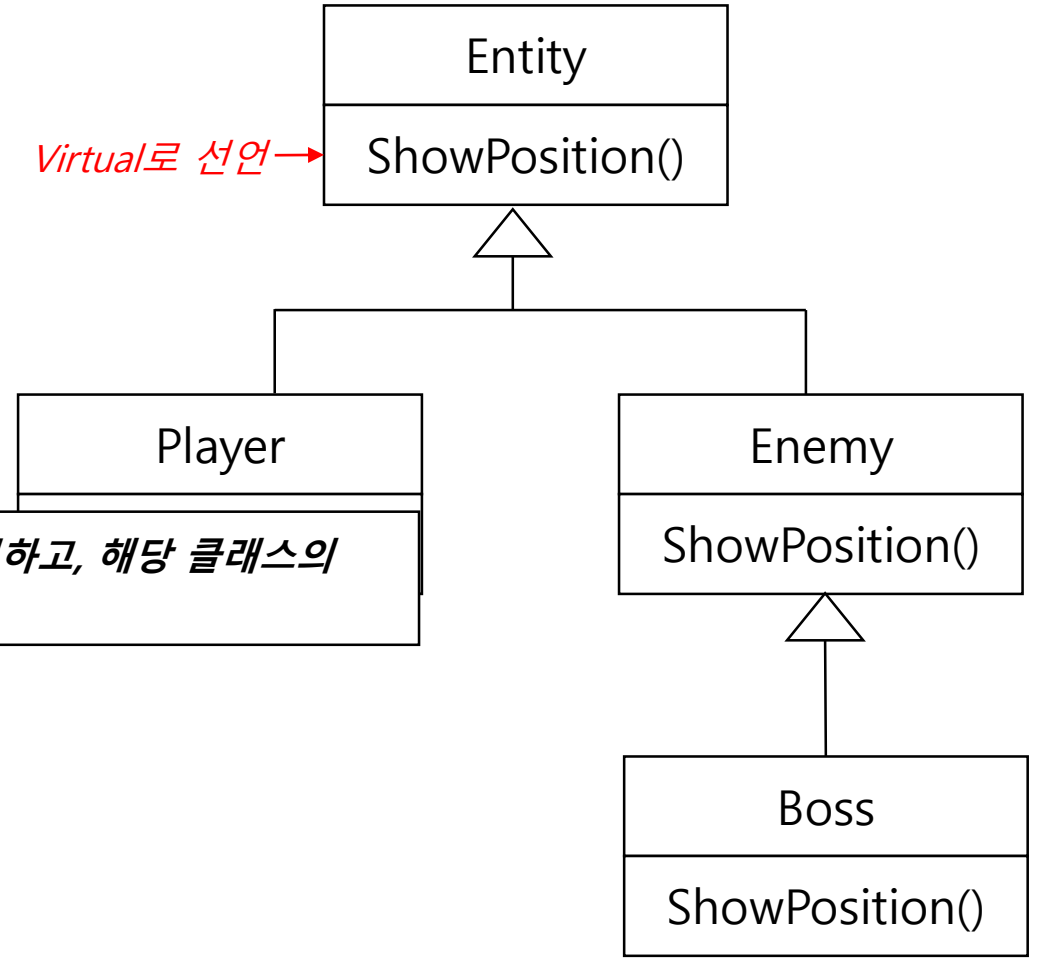


Polymorphism

- 동적 바인딩의 예시 2
  - (동적 바인딩을 구현하는 법은 뒤쪽에서 설명)

```
1  void DisplayPosition(const Entity& e)
2  {
3      e.ShowPosition(); //각자의 ShowPosition()
4  }
5
6  Entity entity{0,0};
7  DisplayPosition(entit
8
9  Player player{0,0,2};
10 DisplayPosition(player);
11
12 Enemy enemy{0,0,2};
13 DisplayPosition(enemy);
```

컴파일러는 런타임에 e의 타입을 확인하고, 해당 클래스의 멤버 함수를 호출



Polymorphism

- 다형성과 동적 바인딩 : 기본 동작인 정적 바인딩은 컴파일시 타입을 기준으로 호출 함수를 결정하지만, 동적 바인딩을 하게되면 런타임시 실제 메모리에 저장된 타입을 기준으로 호출 함수를 결정
- 가상 함수
- 기본 클래스의 포인터 / 참조자
- override/final 지정자
- 순수 가상 함수와 추상 클래스
- 추상 클래스와 인터페이스

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Virtual Function

- 가상 함수
  - Move()와 같이, 유도 클래스에서 기본 클래스의 함수를 재정의 또는 오버라이드해 사용할 수 있음
  - 오버라이드된 함수는 동적 바인딩을 통해 12, 13 페이지의 예제와 같이 활용 가능
  - 오버라이드될 수 있는 함수를 가상함수라 함
    - ➢ 예제에서처럼, 기본 클래스의 함수(Entity::Move)가 가상함수로 선언 후, 유도 클래스에서 해당 함수를 오버라이드해서 구현하면 동적 바인딩됨

- C++에서 런타임 다형성의 구현을 위해 아래와 같은 조건이 필요
  - 상속
  - 기본 클래스 포인터 또는 참조자
  - 가상 함수 ⇦

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Virtual Function

- 가상 함수의 선언, 기본 클래스에서 할일
  - 오버라이드 할 함수를 기본 클래스에서 virtual로 선언 해주어야 함
  - 상속 계층구조에 있는 모든 해당 함수는 가상 함수가 됨

```
1 class Entity{
2 public:
3     virtual void Move(int dx, int dy);
4     ...
5 };
```

기본 클래스의 멤버 함수 앞에 virtual 키워드를 붙이면, 가상 함수가 됨

Virtual Function

중요!

- 가상 함수의 선언, 유도 클래스에서 할일
  - 오버라이드 할 함수를 유도 클래스에서 구현
  - 함수 원형(prototype)과 반환형이 기본 클래스의 가상함수와 일치해야 함!
  - 유도 클래스의 함수에서는 virtual 키워드를 넣지 않아도 되지만, 혼동을 피하기 위해 명시해 주는 것을 권고
  - 유도 클래스에서 함수를 오버라이드하지 않으면, 기존과 같이 기본 클래스의 함수가 상속됨

```
1 class Player : public Entity{
2 public:
3       virtual void Move(int dx, int dy);
4       ...
5 }
```


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Virtual Destructor

중요!

- 가상 소멸자
  - 다형성 객체를 소멸할 때의 고려사항
    - ➢ 포인터를 사용한 뒤 해제할 때, 소멸자가 정적 바인딩되어 있다면 기본 클래스의 소멸자가 호출됨

```cpp
class Entity {
private:
    int x;
    int y;
public:
    Entity(int x, int y)
        : x{ x }, y{ y }    {}
    ~Entity()
    {
        std::cout << "Entity Destructor" << std::endl;
    }
};
class Player : public Entity {
private:
    int hp;
public:
    Player(int x, int y, int hp)
        : Entity{ x,y }, hp{ hp } {}
    ~Player()
    {
        std::cout << "Player Destructor" << std::endl;
    }
};
```

```cpp
int main()
{
    Entity* ptr = new Player{ 2,3,5 };

    //use ptr

    delete ptr;
}
```

ptr을 해제하면 소멸자가 호출됨
*Ptr의 타입은 Entity*이므로 Entity의 소멸자만 호출!*

만일 Player의 멤버에 동적 할당한 포인터가 있었다면 어떤 문제가 발생할까?

강의자료 무
문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Virtual Destructor

중요!

- 가상 소멸자
  - 유도 객체를 올바른 순서로, 올바른 소멸자를 사용해 소멸시키는 방법이 필요
  - 해결 방법? → 가상 소멸자
    - ➢ 클래스가 가상 함수를 가지면, 항상 가상 소멸자를 함께 정의해야 함
    - ➢ 마찬가지로, 기본 클래스의 소멸자가 가상 소멸자면, 유도 클래스의 소멸자도 가상 소멸자

```
1 class Entity{
2 public:
3     virtual ~Entity();
4     ...
5 };
```

앞장의 코드에서 Entity의 소멸자를 가상으로 선언해주면 Player의 소멸자와 Entity의 소멸자가 모두 호출된다.

Player의 소멸자만 호출되는 것이 아닌 이유는?

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Polymorphism

- 다형성과 동적바인딩 : 기본 동작인 정적 바인딩은 컴파일시 타입을 기준으로 호출 함수를 결정하지만, 동적 바인딩을 하게되면 런타임시 실제 메모리에 저장된 타입을 기준으로 호출 함수를 결정
- 가상 함수 : 컴파일러에게 동적 바인딩을 하라고 알려주기 위해서는 기본 클래스의 함수를 virtual 키워드를 통해 가상함수로 만들어주어야 함. 이런 경우, 소멸자도 항상 가상 함수로 선언 필요
- 기본 클래스의 포인터 / 참조자
- override/final 지정자
- 순수 가상 함수와 추상 클래스
- 추상 클래스와 인터페이스

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Base Class Pointer

- 기본 클래스의 포인터

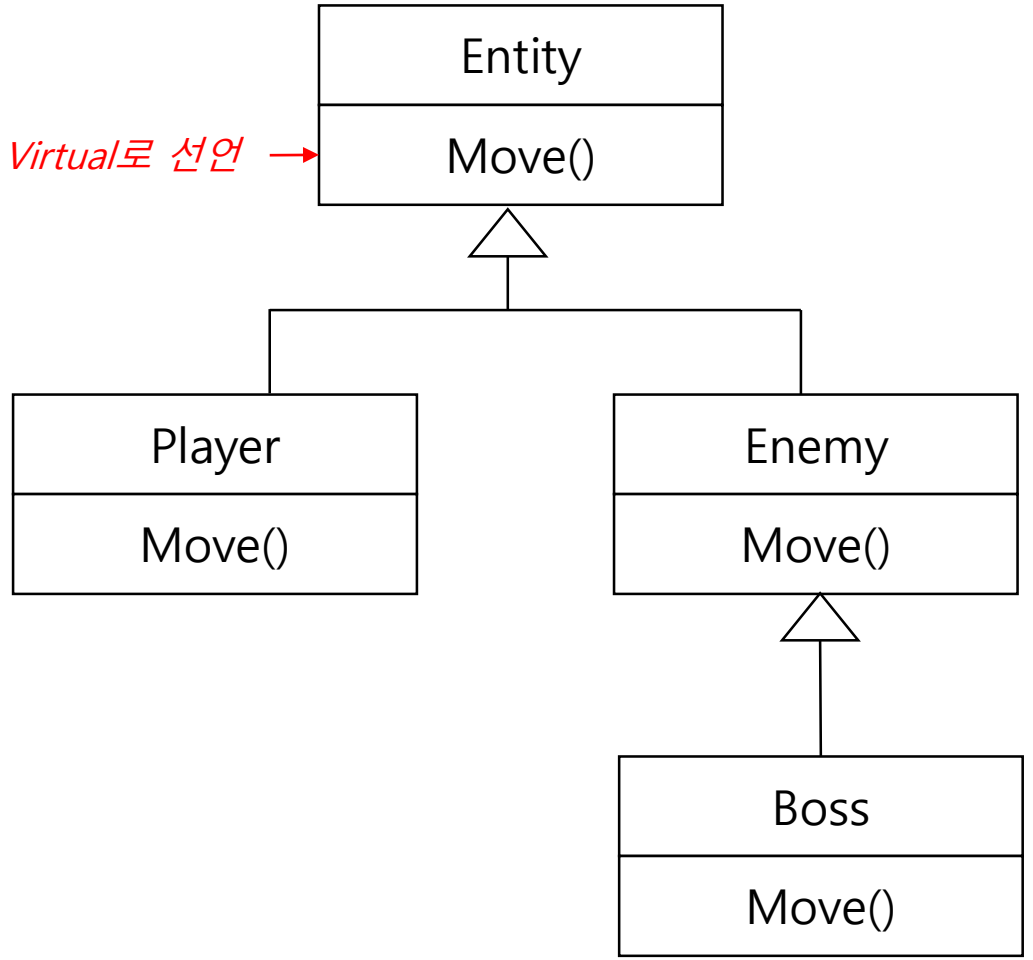
- C++에서 런타임 다형성의 구현을 위해 아래와 같은 조건이 필요
  - 상속
  - 기본 클래스 포인터 또는 참조자 ⇦
  - 가상 함수

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Base Class Pointer

- 기본 클래스의 포인터 사용 예

```
1 Entity *p1 = new Entity{0,0};
2 Entity *p2 = new Player{0,0,2};
3 Entity *p3 = new Enemy{0,0,2};
4 Entity *p4 = new Boss{0,0,2};
5
6 p1→Move(1,1); //Entity::Move()
7 p2→Move(1,1); //Player::Move()
8 p3→Move(1,1); //Enemy::Move()
9 p4→Move(1,1); //Boss::Move()
10
11 //delete pointers
```

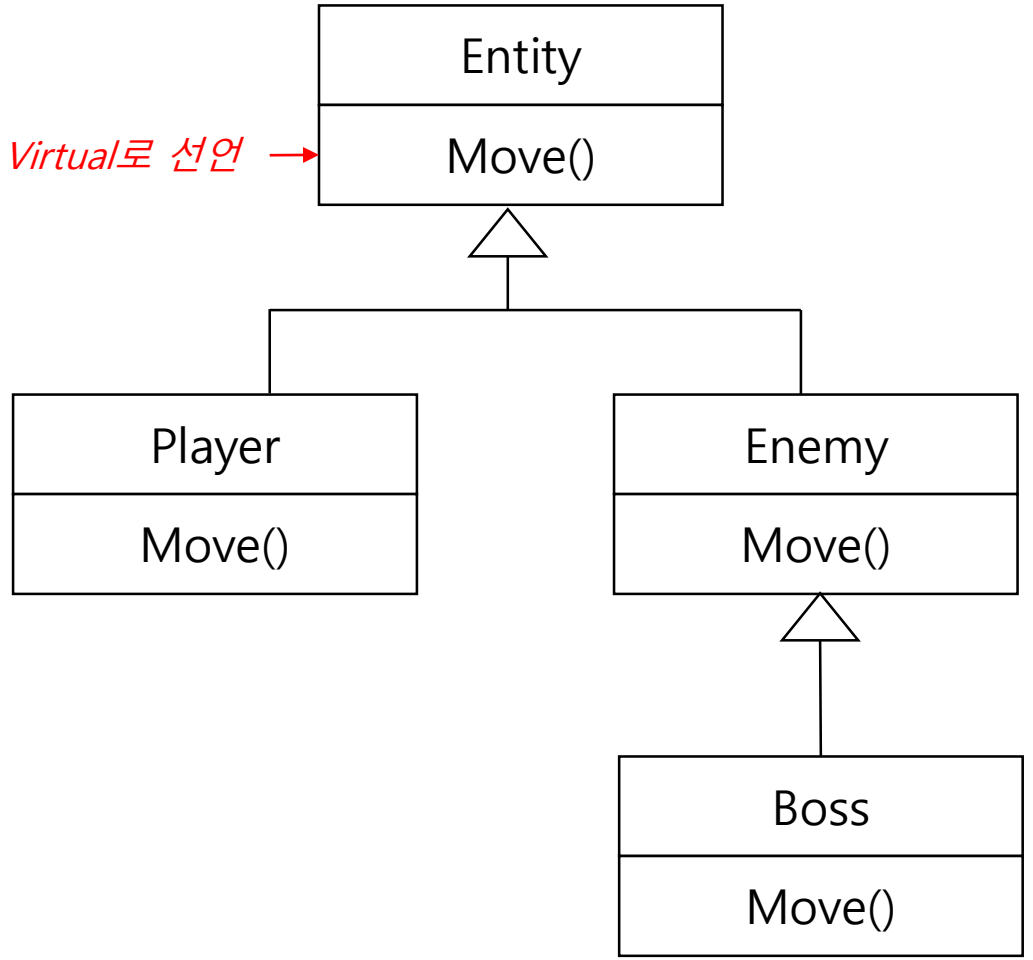


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Base Class Pointer

- 기본 클래스의 포인터 사용 예
  - 아래와 같이 배열을 구성하여 활용 가능!

```
1 Entity *p1 = new Entity{0,0};
2 Entity *p2 = new Player{0,0,2};
3 Entity *p3 = new Enemy{0,0,2};
4 Entity *p4 = new Boss{0,0,2};
5
6 Entity *array[] = {p1,p2,p3,p4};
7
8 for(int i=0;i<4;i++)
9     array[i]→Move(1,1);
```



Using Base Class Reference

- 기본 클래스의 참조자를 활용한 다형성

```cpp
Entity e{0,0};
Entity &ref = e;
ref.Move(1,1); //Entity::Move()

Player p{0,0,2};
Entity &ref2 = p;
ref2.Move(1,1); //Player::Move()
```

```cpp
void MoveX(Entity& e, int dx)
{
    e.Move(dx,0);
}

Entity e{0,0};
MoveX(e,1); //Entity::Move()

Player p{0,0,2};
MoveX(p,1); //Player::Move()
```

**기본 클래스의 참조자이기 때문에 동적 바인딩됨*

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Polymorphism

- 다형성과 동적바인딩 : 기본 동작인 정적 바인딩은 컴파일시 타입을 기준으로 호출 함수를 결정하지만, 동적 바인딩을 하게되면 런타임시 실제 메모리에 저장된 타입을 기준으로 호출 함수를 결정
- 가상 함수 : 컴파일러에게 동적 바인딩을 하라고 알려주기 위해서는 기본 클래스의 해당 함수를 virtual 키워드를 통해 가상함수로 만들어주어야 함. 이런 경우, 소멸자도 항상 가상 함수로 선언 필요
- 기본 클래스의 포인터 / 참조자 : 동적 바인딩을 위해서는 기본 클래스의 포인터/참조자로 유도 클래스를 가리키고 함수를 호출 (가상함수가 아니면 기본 클래스 함수가 호출됨에 주의)
- override/final 지정자
- 순수 가상 함수와 추상 클래스
- 추상 클래스와 인터페이스

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Override Specifier

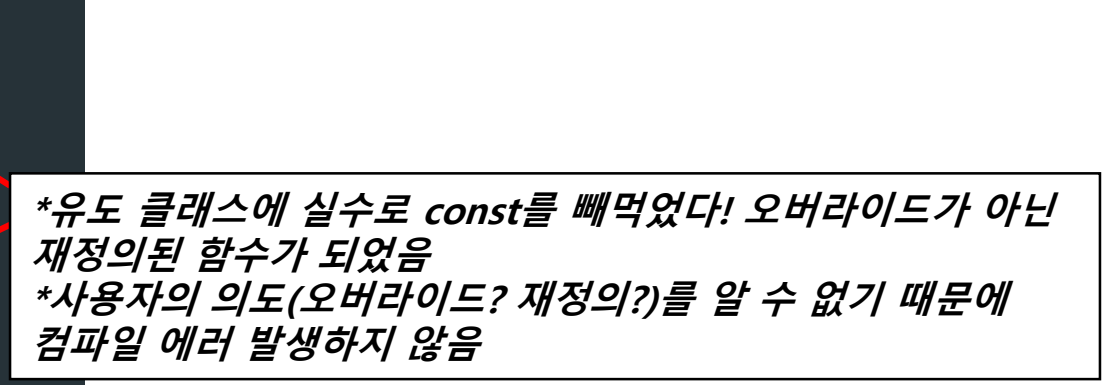
- override 지정자
  - (복습)기본 클래스의 가상 함수는 오버라이드 가능하다.
  - (복습)오버라이드를 위해서는 함수의 원형과 반환형이 동일해야 한다.
    - 만일 다르다면, 오버라이드가 아닌 "재정의" 가 되어버림 (서로 다른 별개의 함수로 인식)
    - 재정의는 정적 바인딩
  - C++ 11부터 override 지정자 기능을 제공하여 오버라이딩시 실수를 방지하고, 코드의 가독성을 상승 가능
    - 어떤 함수가 오버라이딩된 함수인지 정의만 보고도 파악 가능

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

override Specifier

- override 지정자가 필요한 이유

```
1 class Base {
2 public:
3     virtual void sayHello() const {
4         cout << “Hello, I’m Base” << 두이;
5     }
6     virtual ~Base() {}
7 };
8
9 Class Derived : public Base {
10 Public:
11     virtual void sayHello() {
12         cout << "Hello, I'm Derived" << endl;
13     }
14     virtual ~Derived() {}
15 };
```



override Specifier

- override 지정자가 필요한 이유
  - 재정의되었기 때문에, 아래와 같이 의도하지 않게 동작함

```
1 Base *p1 = new Base();
2 p1→sayHello(); //"Hello, I'm Base"
3
4 Base *p2 = new Derived();
5 p2→sayHello(); //"Hello, I'm Base"
```

override Specifier

- override 지정자

```cpp
class Base {
public:
    virtual void sayHello() const {
        cout << "Hello, I'm Base" << endl;
    }
    virtual ~Base() {}
};

class Derived : public Base {
public:
    virtual void sayHello() override {
        cout << "Hello, I'm Derived" << endl;
    }
    virtual ~Derived() {}
};
```

*사용자의 의도(오버라이드)를 파악했기 때문에, 함수 원형이 일치하지 않는다는 컴파일 에러 발생

(p.32의 결과를 보면 실수를 쉽게 알수있지 않나? → 코드의 규모가 커지면 정확한 오류 원인 파악이 어려움. 가능한 빨리 컴파일 단계에서부터 의도하지 않은 동작을 막아야 함)

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

final Specifier

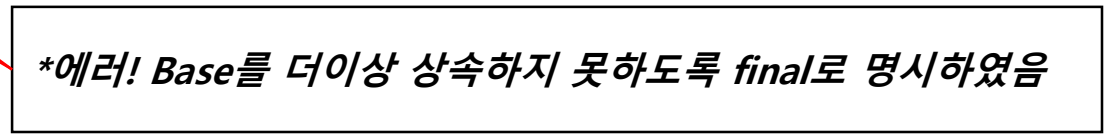
- final 지정자
  - C++ 11부터 final 지정자 기능을 제공
  - 클래스의 final
    - 클래스를 더 이상 상속하지 못하도록 함
  - 멤버 함수의 final
    - 유도 클래스에서 가상 함수를 오버라이드 하지 못하도록 함

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

final Specifier

- final 지정자
  - C++ 11에서는 final 지정자 기능을 제공
  - 클래스의 final

```
1 class Base final {
2
3 };
4
5 class Derived : public Base {
6
7 };
```



final Specifier

- final 지정자
  - C++ 11에서는 final 지정자 기능을 제공
  - 멤버 함수의 final

```
1  class A {
2  public:
3      virtual void doSomething();
4  };
5
6  class B : public A {
7  public:
8      virtual void doSomething() final;
9  };
10
11 class C : public B {
12 public:
13     virtual void doSomething();
14 };
```

B에서의 doSomething 오버라이드는 OK

에러! B에서의 doSomething을 final로 명시하였으므로, C에서는 오버라이드 불가!

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Polymorphism

- 다형성과 동적바인딩 : 정적 바인딩은 컴파일시 타입을 기준으로 호출 함수를 결정하지만, 동적 바인딩은 런타임시 실제 메모리 타입을 기준으로 호출
- 가상 함수 : 컴파일러에게 동적 바인딩을 하라고 알려주기 위해서는 기본 클래스의 해당 함수를 virtual 키워드를 통해 가상함수로 만들어주어야 함. 이런 경우, 소멸자도 항상 가상 함수로 선언 필요
- 기본 클래스의 포인터 / 참조자 : 동적 바인딩을 위해서는 기본 클래스의 포인터/참조자로 유도 클래스를 가리키고 함수를 호출
- override/final 지정자 : override는 가상함수의 오버라이딩을 명시하여 실수 방지, final은 더 이상 상속/오버라이드 하지 못하도록 하는 키워드
- 순수 가상 함수와 추상 클래스
- 추상 클래스와 인터페이스

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Pure Virtual Functions and Abstract Classes

- 추상 클래스 (Abstract Class)
  - 객체를 생성할 수 없는 클래스
  - 상속 계층구조에서 기본 클래스로 사용됨
  - 아주 일반적이어서 객체를 생성하기엔 맞지 않는…
    - Entity (무슨 Entity?), Account (어떤 통장?)
- 구상 클래스 (Concrete Class)
  - 객체를 생성할 수 있는 클래스
  - 모든 멤버 함수가 구현되어 있어야 함
    - 지금까지 예시로 든 모든 클래스는 구상 클래스

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Pure Virtual Functions and Abstract Classes

- 추상 클래스는 하나 이상의 "순수 가상 함수"를 갖는다.
- 즉, "순수 가상 함수"가 있는 클래스는 추상 클래스이다
  - 멤버 함수의 선언 뒤에 "=0"을 붙이면 순수 가상 함수가 됨

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

Pure Virtual Functions and Abstract Classes

중요!

- "순수 가상 함수"가 있는 클래스는 추상 클래스이다.
  - 유도 클래스들은 반드시 기본 클래스의 순수 가상함수를 오버라이드 해야 함
    - ➢ 오버라이드 하지 않는 경우, 유도 클래스도 추상 클래스로 간주됨
    - ➢ 즉, 유도 클래스에서 특정 함수 구현을 "강제" 하는 의미를 가짐
- 사용 목적
  - 기본 클래스에서의 구현이 적절하지 않은 경우
  - 유도 클래스에서는 반드시 구현해야 함을 명시하기 위해
  - Ex) 모든 Entity는 x,y 좌표를 가지고 있고 이동이 가능함. 그러나 실제로 게임 내에서 표현되는 객체가 되려면 어떤 로직으로 이동이 가능한지 구체적 기능이 필요함
    - ➢ 그래서 Player객체, Enemy 객체가 이동이 가능하려면, Move() 함수를 반드시 오버라이딩 해야 함을 강제하기 위해 Move()를 순수 가상 함수로 구현함

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Pure Virtual Functions and Abstract Classes

- 순수 가상 함수와 추상 클래스 예시

```
1 class Shape { //Abstract
2 private:
3     //member variables
4 public:
5     virtual void draw() = 0;
6     virtual void rotate() = 0;
7     virtual ~Shape();
8     ...
9 };
```

*순수 가상 함수들이 있으니, 추상 클래스임

순수 가상 소멸자가 필요할 때도 있을까?

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Pure Virtual Functions and Abstract Classes

- 순수 가상 함수와 추상 클래스 예시

```
1 class Circle : public Shape {
2 private:
3     //member variables for circle
4 public:
5     virtual void draw() override {
6         //implementation for circle
7     }
8     virtual void rotate() override {
9         //implementation for circle
10    }
11    virtual ~Circle();
12    ...
13 };
```

가상 함수들을 모두 오버라이드하여 구현하였으니 구상 클래스가 되었음

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Pure Virtual Functions and Abstract Classes

- 순수 가상 함수와 추상 클래스 예시
  - 추상 클래스는 객체를 생성할 수 없음

```
1 Shape shape; //ERROR!
2 Shape *ptr = new Shape(); //ERROR!
```

  - 하지만 여전히 추상 (기본) 클래스의 포인터/참조자를 사용해 오버라이딩된 함수를 동적 바인딩 할 수 있음

```
1 Shape *ptr = new Circle();
2 ptr→draw();
3 ptr→rotate();
```


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Polymorphism

- 다형성과 동적바인딩 : 정적 바인딩은 컴파일시 타입을 기준으로 호출 함수를 결정하지만, 동적 바인딩은 런타임시 실제 메모리 타입을 기준으로 호출
- 가상 함수 : 컴파일러에게 동적 바인딩을 하라고 알려주기 위해서는 기본 클래스의 해당 함수를 virtual 키워드를 통해 가상함수로 만들어주어야 함. 이런 경우, 소멸자도 항상 가상 함수로 선언 필요
- 기본 클래스의 포인터 / 참조자 : 동적 바인딩을 위해서는 기본 클래스의 포인터/참조자로 유도 클래스를 가리키고 함수를 호출
- override/final 지정자 : override는 가상함수의 오버라이딩을 명시하여 실수 방지, final은 더 이상 상속/오버라이드 하지 못하도록 하는 키워드
- 순수 가상 함수와 추상 클래스 : 순수 가상함수가 있다면 추상 클래스이고, 이런 클래스는 객체를 생성할 수 없음. 유도 클래스에서 순수 가상함수를 오버라이딩 해야 객체 생성 가능
- 추상 클래스와 인터페이스

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Abstract Classes and Interface

중요!

- 추상 클래스를 사용한 인터페이스 클래스
  - 순수 가상 함수"만"을 가진 추상 클래스를 인터페이스 클래스라 함
    - (C#, JAVA 언어는 따로 interface라는 키워드로 인터페이스 구현이 가능)
  - 클래스의 사용에 있어서 일반적인 기능(서비스)을 묶어놓은 클래스
  - 인터페이스 클래스를 기반으로 한 구상 클래스는 모든 기능(함수/서비스)를 구현해야 함
  - 인터페이스는 껍데기(?)다
  - 유도 클래스가 꼭 가져야 하는 기능들을 명시해 놓기 위해서 사용

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Abstract Classes and Interface

- 추상 클래스를 사용한 인터페이스 클래스 예시

```
1  class Shape { //Abstract, Interface
2  public:
3      virtual void draw() = 0;
4      virtual void rotate() = 0;
5      virtual ~Shape();
6  };
7
8  class Circle : public Shape {
9  public:
10     virtual void draw() override {
11         //implementation for circle
12     }
13     virtual void rotate() override {
14         //implementation for circle
15     }
16     virtual ~Circle();
17     ...
18 };
```

순수 가상함수만을 가지고 있는 Shape 인터페이스

Shape 인터페이스 클래스를 상속한 Circle 구상 클래스

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Abstract Classes and Interface

- 추상 클래스를 사용한 인터페이스 클래스 예시
  - 통상적으로 인터페이스 클래스의 이름을 지을 땐 대문자 I를 앞에 붙임

```
1 class IShape { //Abstract, Interface
2 public:
3     virtual void draw() = 0;
4     virtual void rotate() = 0;
5     virtual ~IShape();
6     ...
7 };
```

추가 슬라이드

vtable

- 이전에 본 것처럼, 클래스 객체가 저장된 메모리에 멤버 함수 구현이 포함되어 있지 않음
  - 구현된 멤버 함수는 메모리의 코드 영역에 들어있음
- 클래스가 가상 함수를 가지고 있는 경우, 컴파일러는 오버라이딩에 따르는 함수의 호출 정보가 명시된 가상함수 테이블(vtable)을 가리키는 포인터(vptr)를 객체에 포함시킴

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

vtable

```
class AAA {
private:
        int num1;
public:
        virtual void Func1()
        {
                cout << "func1" << endl;
        }
        virtual void Func2()
        {
                cout << "func2" << endl;
        }
};

class BBB : public AAA {
private:
        int num2;
public:
        virtual void Func1() override
        {
                cout << "BBB::func1" << endl;
        }
        void Func3()
        {
                cout << "func3" << endl;
        }
};
```

```
int main()
{
        AAA a;
        BBB b;

        cout << sizeof(a) << endl; //vtable을 가리키는 포인터로 인해 4 byte 추가 메모리
        cout << sizeof(b) << endl; //vtable을 가리키는 포인터로 인해 4 byte 추가 메모리

}
```

vtable

가상함수의 동작원리와 가상함수 테이블

윤성우의 열혈 C++ 프로그래밍

```
class AAA
{
private:
    int num1;
public:
    virtual void Func1() { cout<<"Func1"<<endl; }
    virtual void Func2() { cout<<"Func2"<<endl; }
};
class BBB: public AAA
{
private:
    int num2;
public:
    virtual void Func1() { cout<<"BBB::Func1"<<endl; }
    void Func3() { cout<<"Func3"<<endl; }
};
int main(void)
{
    AAA * aptr=new AAA();
    aptr->Func1();
    BBB * bptr=new BBB();
    bptr->Func1();
    return 0;
}
```

```
Func1
BBB::Func1
```

실행결과

key	value
void AAA::Func1( )	0x1024 번지
void AAA::Func2( )	0x2048 번지

▶ [그림 09-3: AAA 클래스의 가상함수 테이블]

key	value
void BBB::Func1( )	0x3072 번지
void AAA::Func2( )	0x2048 번지
void BBB::Func3( )	0x4096 번지

▶ [그림 09-4: BBB 클래스의 가상함수 테이블]

하나 이상의 가상함수가 멤버로 포함되면 위와 같은 형태의 V-Table이 생성되고 매 함수호출시마다 이를 참조하게 된다.

BBB 클래스의 가상함수 테이블에는 AAA::Func1에 대한 정보가 없음에 주목하자!

vtable

가상함수 테이블이 참조되는 방식

윤성우의 열혈 C++ 프로그래밍