



# C++ 프로그래밍

김 형 기

[hk.kim@jbnu.ac.kr](mailto:hk.kim@jbnu.ac.kr)

상속

# Example

- 상속이 필요한 이유
  - Player 클래스

```
1 class Player
2 {
3 private:
4     int x, y;
5     int speed;
6 public:
7     Player(int x, int y, int speed)
8         : x{ x }, y{ y }, speed{ speed }
9     {}
10    void Move(int dx, int dy)
11    {
12        x += dx * speed;
13        y += dy * speed;
14    }
15    void ShowPosition()
16    {
17        cout << x << "," << y << endl;
18    }
19 };
```

# Example

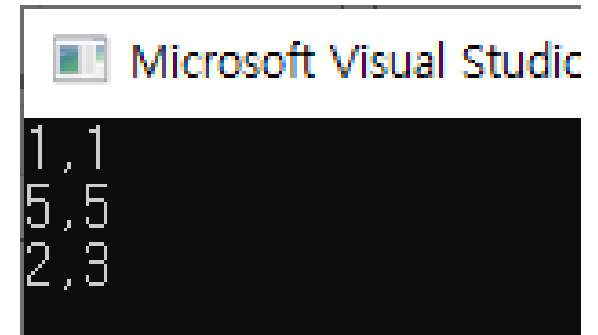
- 상속이 필요한 이유
  - Player 관리를 위한 클래스
    - 컨트롤/매니저 클래스라 부름

```
1 class PlayerHandler
2 {
3 private:
4     Player* playerList[50];
5     int playerNum;
6 public:
7     PlayerHandler() : playerNum{ 0 } {}
8     void AddPlayer(Player* p)
9     {
10         playerList[playerNum++] = p;
11     }
12     void ShowAllPlayerPosition() const
13     {
14         for (int i = 0; i < playerNum; i++)
15         {
16             playerList[i]→ShowPosition();
17         }
18     }
19     ~PlayerHandler()
20     {
21         for (int i = 0; i < playerNum; i++)
22         {
23             delete playerList[i];
24         }
25     }
26 };
```

# Example

- 상속이 필요한 이유
  - main함수

```
1 int main()
2 {
3     PlayerHandler playerHandler;
4     playerHandler.AddPlayer(new Player(1, 1, 1));
5     playerHandler.AddPlayer(new Player(5, 5, 1));
6     playerHandler.AddPlayer(new Player(2, 3, 1));
7     playerHandler.ShowAllPlayerPosition();
8 }
```



# Example

---

- 상속이 필요한 이유, 시나리오
  - Player 이외에 Enemy와 NPC가 추가된다면?
  - Enemy와 NPC의 이동 방식이 다르다면?
    - Enemy :  $dx * speed * 1.5f$ ;
    - NPC : 이동 불가능
  - Enemy 및 NPC 클래스를 추가 구현했을 때, 컨트롤 클래스를 얼마나 수정해야 하는가?

# Example

- 상속이 필요한 이유, 시나리오 예상

- 멤버 변수 추가 필요
- 클래스별 정보 추가  
기능 필요
- 클래스별 위치 정보  
출력 반복문 필요
- 클래스별 해제 필요  
등등...

→ 상속과 다형성을 활용한다면 적은 수정으로 기능  
추가 가능하도록 설계가 가능!

```
1 class PlayerHandler
2 {
3 private:
4     Player* playerList[50];
5     int playerNum;
6 public:
7     PlayerHandler() : playerNum{ 0 } {}
8     void AddPlayer(Player* p)
9     {
10         playerList[playerNum++] = p;
11     }
12     void ShowAllPlayerPosition() const
13     {
14         for (int i = 0; i < playerNum; i++)
15         {
16             playerList[i]→ShowPosition();
17         }
18     }
19     ~PlayerHandler()
20     {
21         for (int i = 0; i < playerNum; i++)
22         {
23             delete playerList[i];
24         }
25     }
26 };
```

# Inheritance

---

- 상속의 정의
- 유도 클래스
- protected 멤버
- 상속에서의 생성자와 소멸자
- 기본 클래스의 생성자와의 관계
- 상속과 멤버 함수



# Inheritance

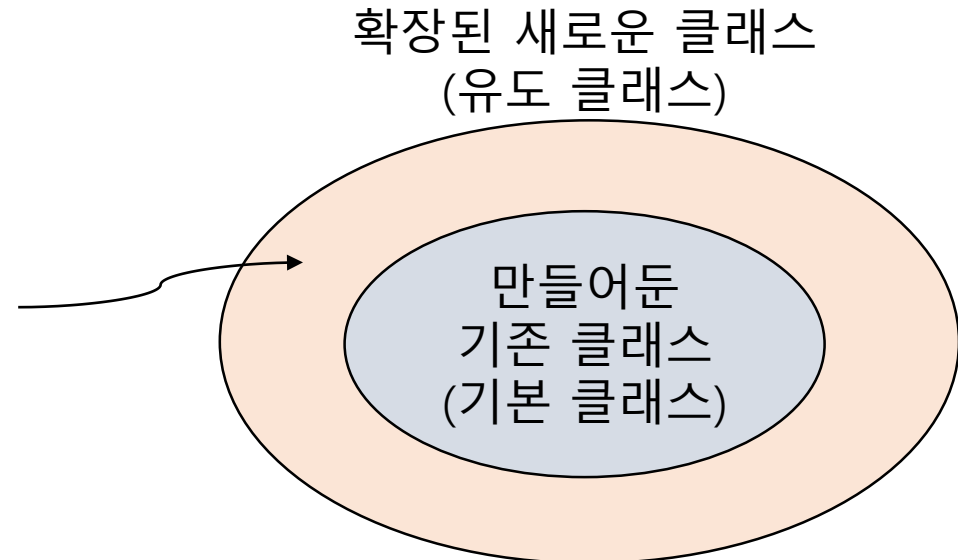
---

- 상속의 정의
- 유도 클래스
- protected 멤버
- 상속에서의 생성자와 소멸자
- 기본 클래스의 생성자와의 관계
- 상속과 멤버 함수

### ● 상속

- 기존 클래스를 활용하여 새로운 클래스를 생성하는 방법
- 새로운 클래스는 **기존 클래스의 데이터와 행동(함수)를 포함**
- 기존 클래스를 **재사용** 가능하게 함
- 클래스들 간의 공통 속성에 집중하는 설계 방법
- 기존 클래스의 행동(함수)을 수정하여, 새로운 클래스의 행동을 새로 정의 가능
  - 기존 클래스의 행동을 수정할 필요는 없음

새로운 기능과 데이터를 추가하여 확장.  
이때 이미 만들어진 클래스의 기능과  
데이터는 모두 포함됨



# Inheritance

---

- 관련된 클래스의 예시

- Player, Enemy, NPC, Boss, Hero, Super Hero, etc.
- Account, Saving Account, Checking Account, Trust Account, etc.
- Shape, Line, Oval, Circle, Square, etc.
- Person, Employee, Student, Faculty, Staff, Administrator, etc.

# Inheritance

## ● Player 예시 설계

### ■ Player

➤ x, y, speed, hp, xp, Talk(), Move()

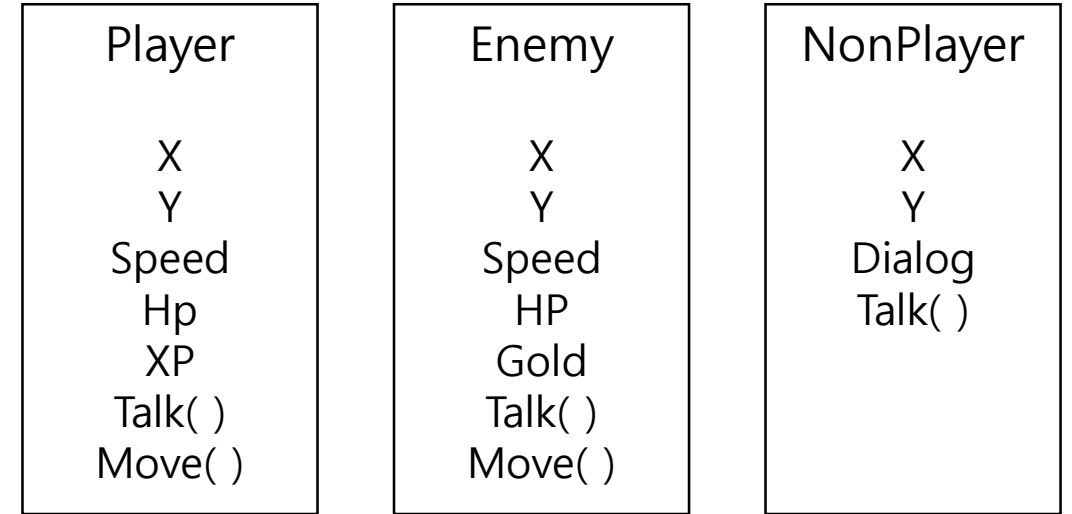
### ■ Enemy

➤ x, y, speed, hp, gold, Talk(), Move()

### ■ NonPlayer

➤ x, y, dialog, Talk()

### ■ 행동(함수)은 클래스마다 다를 수 있음

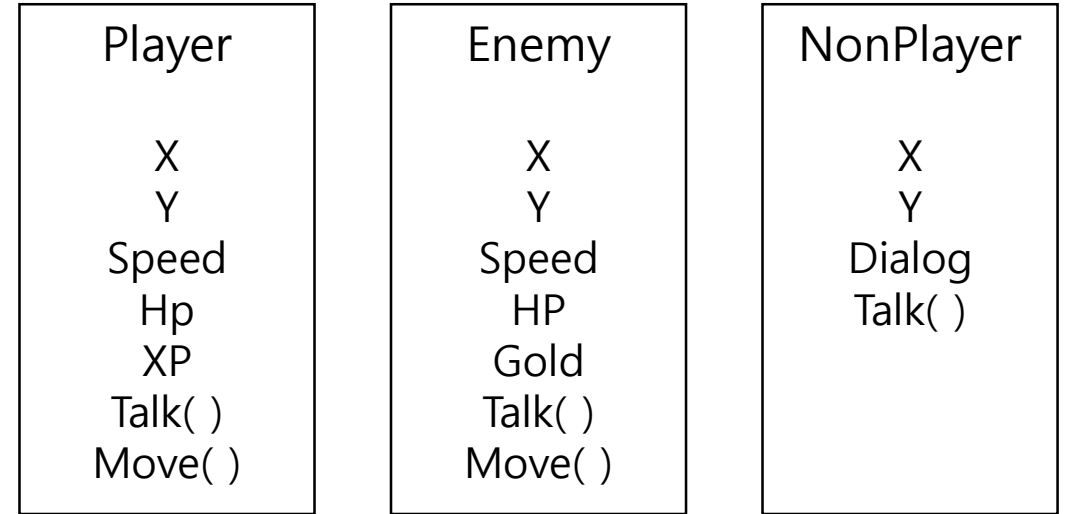


# Inheritance

## ● Player 예시 설계

- 상속을 사용하지 않은 예 (코드의 중복 작성)

```
1 class Player {  
2     //x, y, speed, hp, xp, ...  
3 };  
4  
5 class Enemy {  
6     //x, y, speed, xp, gold...  
7 };  
8  
9 class NonPlayer {  
10    //x, y, dialog, ...  
11 };
```



# Inheritance

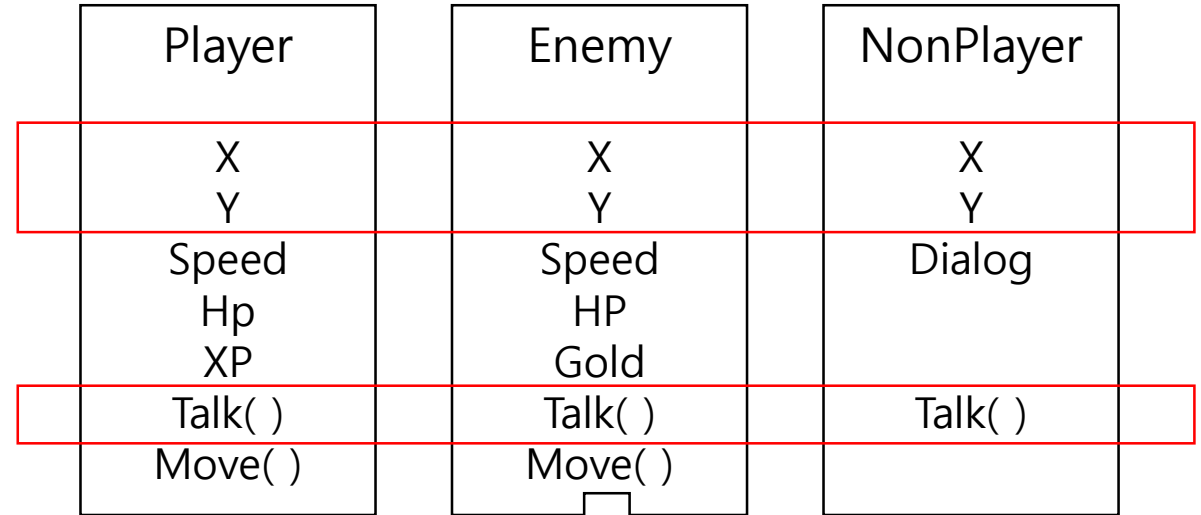
## ● Player 예시 설계

- 상속을 사용한 예 (코드의 재사용)

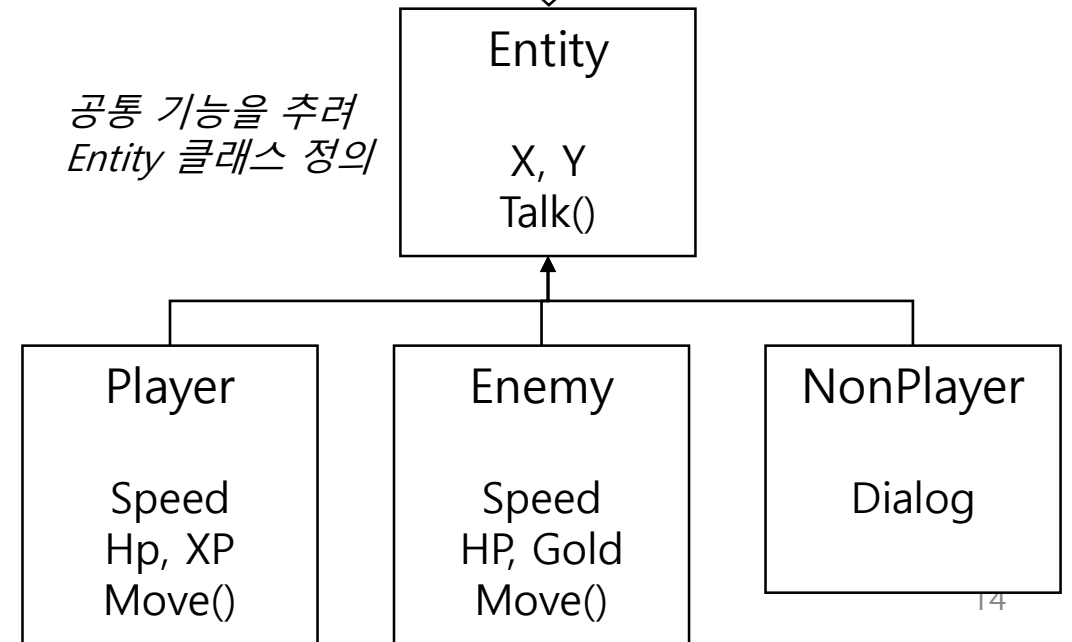
```
1 class Entity {  
2     //x, y, Talk()  
3 };  
4  
5 class Player : public Entity{  
6     //speed, hp, xp, Move()  
7 };  
8  
9 class Enemy : public Entity {  
10    //speed, hp, gold, Move()  
11 };  
12  
13 class NonPlayer : public Entity {  
14    //dialog  
15 };
```

공통 데이터

공통 기능



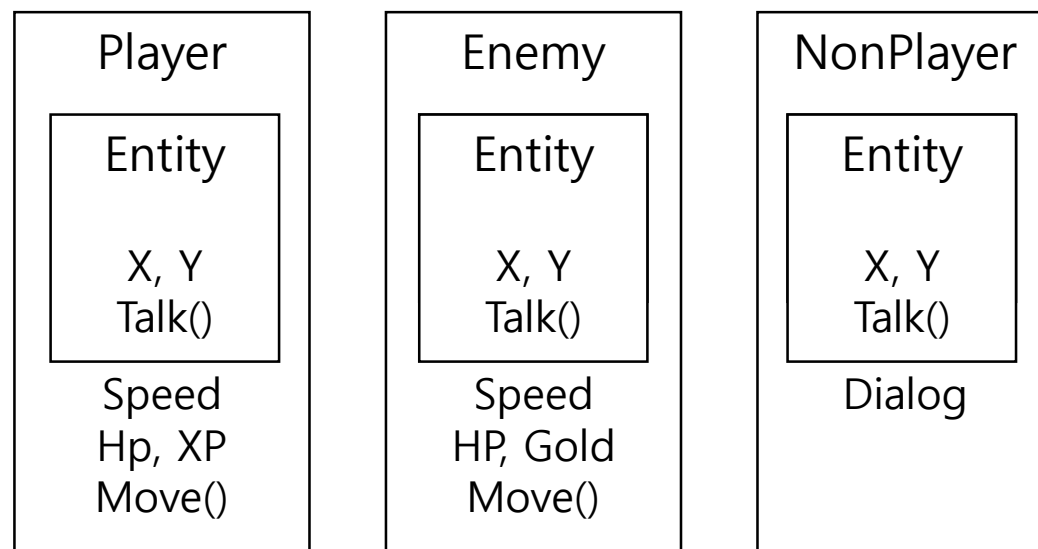
공통 기능을 추려  
Entity 클래스 정의



Entity를 확장하여 Player,  
Enemy, NonPlayer 클래스 정의

### ● Player 예시 설계

- 상속을 사용한 예 (코드의 재사용)
- 우측과 같이 Entity의 모든 데이터와 기능이 Entity를 상속한 클래스들에 포함됨
- Player 객체는 두 가지 타입을 모두 가진 상태가 됨
  - i.e. Player 객체는 Player 타입이기도 하지만 Entity 타입이기도 함!
  - 강의 마지막 Is-A 관계 부분에서 다시 설명



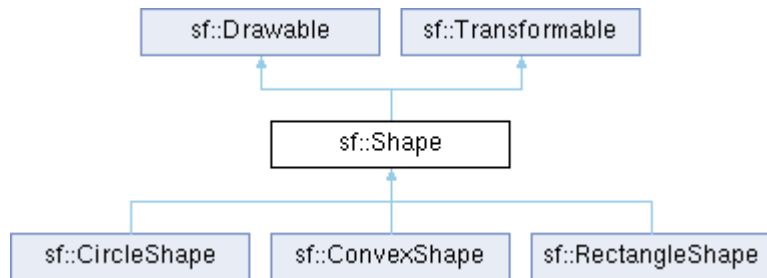
# Terminology

- 기본 클래스 (base class/parent class/super class)

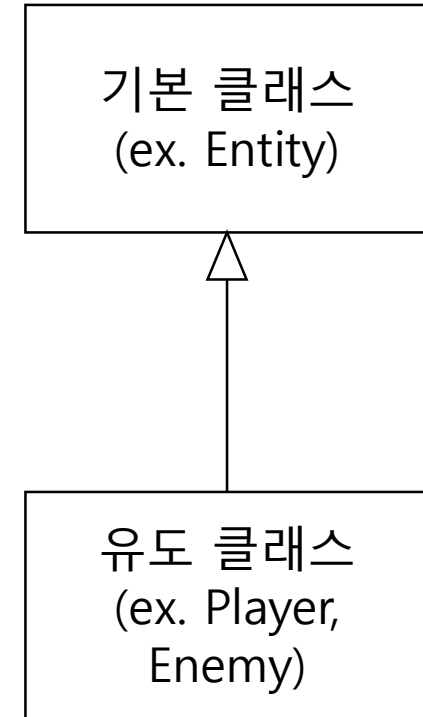
- 상속의 대상이 되는 클래스

- 유도 클래스 (derived class/child class/sub class)

- 기본 클래스로부터 생성되는 클래스
- 데이터와 행동을 기본 클래스로부터 상속함



SFML 라이브러리 문서에 나타난 상속관계.  
sf::Shape 기본 클래스를 바탕으로  
sf::CircleShape, sf::RectangleShape 등 유도  
클래스가 정의되었음을 나타냄



UML 표현 형식으로 나타낸 상속 관계



# Inheritance

---

- 상속의 정의 : 기본 클래스를 기반으로 새 클래스를 만드는 기능
- 유도 클래스
- protected 멤버
- 상속에서의 생성자와 소멸자
- 기본 클래스의 생성자와의 관계
- 상속과 멤버 함수

- C++ 상속 문법

```
1 class Base{
2     //base class members...
3 };
4
5 class Derived : public Base{
6     //derived class members...
7 };
```

*\*Public 위치에 private, protected 키워드를 넣게 되면 다른 방식으로 동작하지만, 잘 사용하지 않으므로 설명하지 않습니다.*

- 예시

```
1 class Entity {
2     //base class members...
3 };
4
5 class Player : public Entity{
6     //derived class members...
7 };
```

# Deriving classes from existing class

---

- Public 상속

- 가장 흔히 사용되는 상속 방식
- "is-a" 관계의 정의와 가장 일치하는 상속 방식

- Private과 protected 상속

- "has-a"와 유사한 관계를 정의하는 상속
- 소유와는 차이가 존재

- 본 강의에서는 public 상속에 집중

- 타 언어에서 protected/private 상속을 아예 지원하지 않는 경우가 많음
- Protected/private 상속은 public 상속만큼 활용도가 높지 않음 (다른 방식으로 구현함)

# Deriving classes from existing class

## ● Player 예제

```
1 class Entity {
2 protected:
3     int x, y;
4 public:
5     Entity(int x, int y)
6         : x{ x }, y{ y }
7     {}
8     void ShowPosition()
9     {
10         cout << x << "," << y << endl;
11     }
12     void Talk()
13     {
14         cout << "안녕하세요." << endl;
15     }
16 };
17 class Player : public Entity {
18 private:
19     int hp, xp, speed;
20 public:
21     Player(int x, int y, int speed)
22         : Entity{ x,y }, speed{ speed }
23     {}
24     void Move(int dx, int dy)
25     {
26         x += dx;
27         y += dy;
28     }
29 };
```

```
1 int main()
2 {
3     Entity entity1{ 1,1 };
4     entity1.Talk();
5     entity1.ShowPosition();
6
7     Player player1{ 2,3,1 };
8     player1.Talk();
9     player1.ShowPosition();
10
11 }
```

*Player에는 x,y도 없고, Talk(), ShowPosition()도 없어 보이지만,*

*x,y,speed와 Talk(), ShowPosition을 모두 포함하고 있는 것을 알 수 있음*

# Deriving classes from existing class

## ● Player 예제

```
1 class Entity {
2 protected:
3     int x, y;
4 public:
5     Entity(int x, int y)
6         : x{ x }, y{ y }
7     {}
8     void ShowPosition()
9     {
10         cout << x << "," << y << endl;
11     }
12     void Talk()
13     {
14         cout << "안녕하세요." << endl;
15     }
16 };
17 class Player : public Entity {
18 private:
19     int hp, xp, speed;
20 public:
21     Player(int x, int y, int speed)
22         : Entity{ x,y }, speed{ speed }
23     {}
24     void Move(int dx, int dy)
25     {
26         x += dx;
27         y += dy;
28     }
29 };
```

### Entity

x,y

Talk()  
ShowPosition()

### Player

#### Entity

X,y

Talk()  
ShowPosition()

speed  
hp, xp  
Move()

# Inheritance

---

- 상속의 정의 : 기본 클래스를 기반으로 새 클래스 만드는 기능
- 유도 클래스 : 그렇게 만들어진 클래스를 유도 클래스라 하며, 기본 클래스의 모든 것이 포함되어 있음
- protected 멤버
- 상속에서의 생성자와 소멸자
- 기본 클래스의 생성자와의 관계
- 상속과 멤버 함수

# Protected Member

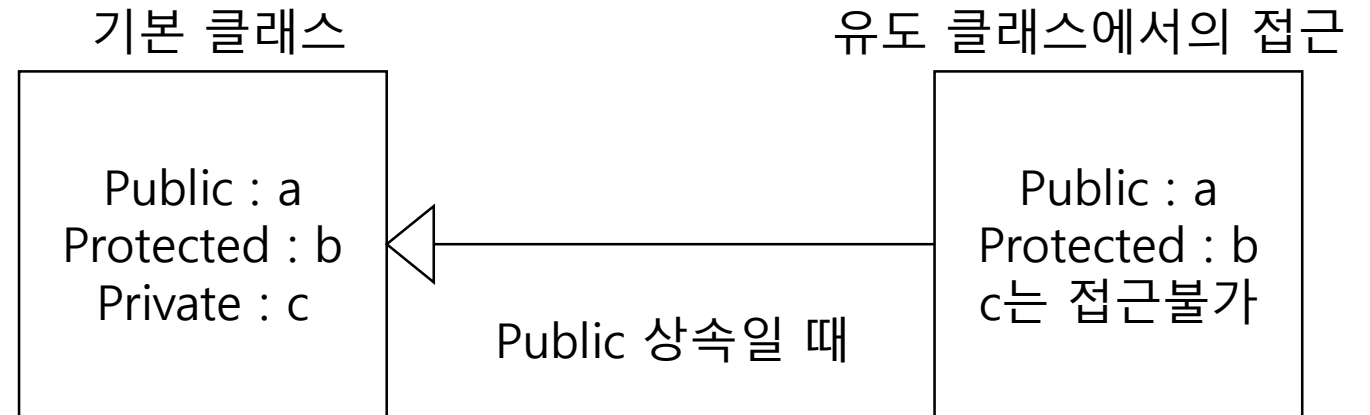
- protected 멤버

- 기본 클래스에서 접근 가능
- 유도 클래스에서 접근 가능
- 기본 또는 유도 클래스의 객체로부터는 접근 불가능!
  - Private을 생각해보자. 클래스에서는 접근 가능하고, 객체로부터는 접근이 불가능하였음
  - Protected는 "상속이 이루어지는 private"라고 생각하면 편함
  - Private은 유도 클래스에서 접근이 불가능함에 유의
    - private은 상속과 관계없이 무조건 (자신) 클래스 내부에서만 접근 가능함

```
1 class Base
2 {
3     protected:
4         //protected members
5 };
```

- protected 멤버 변수

```
1 class Base
2 {
3 public:
4     int a;
5
6 protected:
7     int b;
8
9 private :
10     int c;
11 };
```





# Protected Member

- protected 멤버 변수

```
1 class Base
2 {
3 public:
4     int a;
5
6 protected:
7     int b;
8
9 private :
10     int c;
11 };
```

```
1 class Derived : public Base
2 {
3 public:
4     void SetValue()
5     {
6         cout << a << endl; // OK!
7         cout << b << endl; // OK!
8         cout << c << endl; // ERROR!
9     }
10 };
11
12 int main()
13 {
14     Derived d;
15     d.a = 1; // OK!
16     d.b = 2; // ERROR!
17     d.c = 3; // ERROR!
18
19     cout << sizeof(Base) << endl; // 12 Bytes
20     cout << sizeof(Derived) << endl; // 12 Bytes
21 }
```

접근할 수 없는 것이지, 값이 메모리에  
저장되지 않는 것은 아님

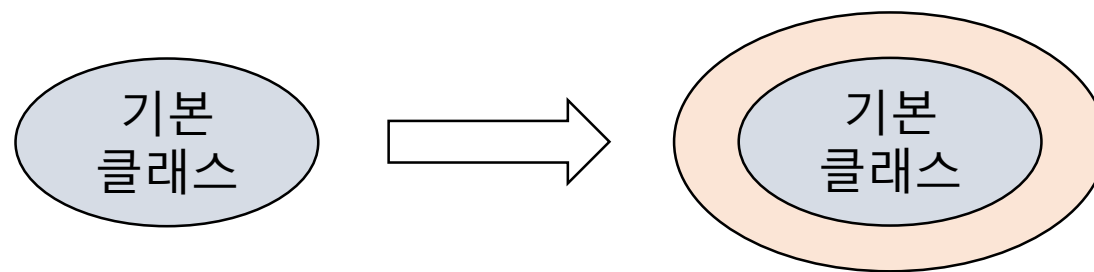
# Inheritance

---

- 상속의 정의 : 기본 클래스를 기반으로 새 클래스 만드는 기능
- 유도 클래스 : 그렇게 만들어진 클래스를 유도 클래스라 하며, 기본 클래스의 모든 것이 포함되어 있음
- protected 멤버 : 상속에서 public은 그대로 public, private은 여전히 private이므로, 정보를 숨기되 상속 계층의 하위로 데이터를 상속하고 싶을 때는 protected를 사용
- 상속에서의 생성자와 소멸자
- 기본 클래스의 생성자와의 관계
- 상속과 멤버 함수

### ● 상속에서의 생성자

- 유도 클래스는 기본 클래스의 멤버를 포함하므로, 유도 클래스가 초기화 되기 **이전에** 기본 클래스에서 상속된 부분이 반드시 초기화 되어야 함
- 유도 클래스 객체가 생성될 때,
  - 먼저 기본 클래스의 생성자가 호출되고,
  - 그 이후 유도 클래스의 생성자가 호출됨



안쪽(기본 클래스)을 먼저 만든 뒤,

바깥쪽(유도 클래스)이 만들어집니다.

# Constructors and Destructors

- 상속에서의 생성자

```
1 class Base
2 {
3 public:
4     Base() { cout << "Base constructor" << endl; }
5 };
6
7 class Derived : public Base
8 {
9 public:
10     Derived() { cout << "Derived constructor" << endl; }
11 };
```

# Constructors and Destructors

- 상속에서의 생성자

```
1 class Base
2 {
3 public:
4     Base() { cout << "Base constructor" << endl; }
5 };
6
7 class Derived : public Base
8 {
9 public:
10     Derived() { cout << "Derived constructor" << endl; }
11 };
```

2) 기본 클래스의 생성자가 호출된 후

3) 유도 클래스의 생성자가 호출된다.

```
1 int main() {
2     Derived derived;
3 }
```

1) 유도 클래스 객체를 생성하면,

# Constructors and Destructors

- 상속에서의 생성자

```
#include<iostream>

using namespace std;

class Base
{
public:
    Base() { cout << "Base constructor" << endl; }
};

class Derived : public Base
{
public:
    Derived() { cout << "Derived constructor" << endl; }
};

int main()
{
    Derived derived;

    return 0;
}
```

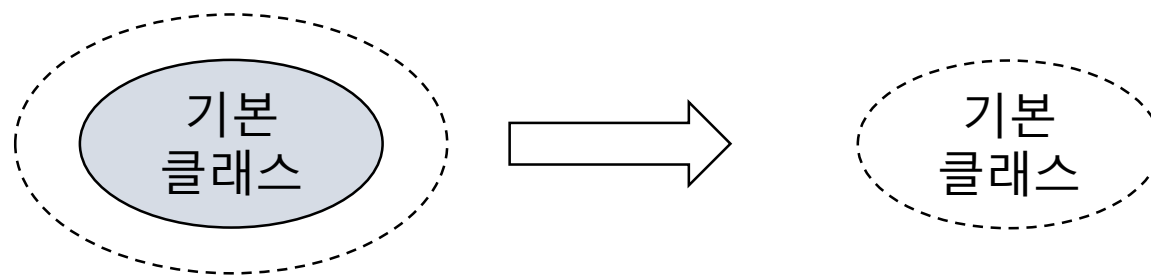
Microsoft Visual Studio 디버그 콘솔

Base constructor  
Derived constructor

E:\tmp\vs\Test\Debug\Test.exe(20432 프로세스)이(가) 0 코드로 인해 종료되었습니다.  
이 창을 닫으려면 아무 키나 누르세요.

- 상속에서의 소멸자

- 소멸자는 생성자와 반대 순서로 호출됨
- 즉, 유도 클래스가 소멸될 때,
  - 먼저 유도 클래스의 소멸자가 호출되고,
  - 그 이후 기본 클래스의 소멸자가 호출됨



바깥쪽(유도 클래스)이 먼저 소멸되고,

안쪽(기본 클래스)이 소멸됩니다.

# Constructors and Destructors

- 상속에서의 소멸자

```
1 class Base
2 {
3 public:
4     Base() { cout << "Base constructor" << endl; }
5     ~Base() { cout << "Base destructor" << endl; }
6 };
7
8 class Derived : public Base
9 {
10 public:
11     Derived() { cout << "Derived constructor" << endl; }
12     ~Derived() { cout << "Derived destructor" << endl; }
13 };
```



# Constructors and Destructors

- 상속에서의 소멸자

```
#include<iostream>

using namespace std;

class Base
{
public:
    Base() { cout << "Base constructor" << endl; }
    ~Base() { cout << "Base destructor" << endl; }
};

class Derived : public Base
{
public:
    Derived() { cout << "Derived constructor" << endl; }
    ~Derived() { cout << "Derived destructor" << endl; }
};

int main()
{
    Derived derived;

    return 0;
}
```



Microsoft Visual Studio 디버그 콘솔

```
Base constructor
Derived constructor
Derived destructor
Base destructor
```

E:\http\vs\Test\Debug\Test.exe(4920 프로세스)  
이 창을 닫으려면 아무 키나 누르세요.

# Constructors and Destructors

---

- 생성자와 소멸자의 상속

- 유도 클래스는 기본 클래스의 생성자, 소멸자 및 오버로딩된 대입 연산자를 상속하지 않음
- 대신, 기본 클래스의 생성자, 소멸자 및 오버로딩된 대입 연산자를 유도 클래스로부터 호출이 가능함
  - 다음 페이지에서 설명
- 따라서, 생성자 오버로딩이 되어 있는 경우 어떤 생성자를 사용할 지 명시해 주는 것이 좋음

# Constructors and Destructors

```
1 class Base {
2 private:
3     int value;
4 public:
5     Base() : value{0} { cout << "Base no-args constructor" << endl; }
6     Base(int x) : value{x} { cout << "Base (int) overloaded constructor" << endl; }
7     ~Base(){ cout << "Base destructor" << endl; }
8 };
9
10 class Derived : public Base {
11 private:
12     int doubled_value;
13 public:
14     Derived() : doubled_value {0} { cout << "Derived no-args constructor " << endl; }
15     Derived(int x) : doubled_value {x*2} { cout << "Derived (int) overloaded constructor" << endl; }
16     ~Derived() { cout << "Derived destructor " << endl; }
17 };
```

```
1 class Base {
2 private:
3     int value;
4 public:
5     Base() : value{0} { cout << "Base no-args constructor" << endl; }
6     Base(int x) : value{x} { cout << "Base (int) overloaded constructor" << endl; }
7     ~Base(){ cout << "Base destructor" << endl; }
8 };
9
10 class Derived : public Base {
11 private:
12     int doubled_value;
13 public:
14     Derived() : doubled_value {0} { cout << "Derived no-args constructor " << endl; }
15     Derived(int x) : doubled_value {x*2} { cout << "Derived (int) overloaded constructor" << endl; }
16     ~Derived() { cout << "Derived destructor " << endl; }
17 };
```

```
1 int main() {
2     Base b; //value: 0
3     Base b{100}; //value: 100
4     Derived d; //value: 0, doubled_value: 0
5     Derived d {1000}; //value: 0, doubled_value: 2000
6 }
```

# Inheritance

---

- 상속의 정의 : 기본 클래스를 기반으로 새 클래스 만드는 기능
- 유도 클래스 : 그렇게 만들어진 클래스를 유도 클래스라 하며, 기본 클래스의 모든 것이 포함되어 있음
- protected 멤버 : 상속에서 public은 그대로 public, private은 여전히 private이므로, 정보를 숨기되 상속 계층의 하위로 데이터를 상속하고 싶을 때는 protected를 사용
- 상속에서의 생성자와 소멸자 : 기본 생성-유도 생성 / 유도 소멸-기본 소멸 순서
- 기본 클래스 생성자와의 관계
- 상속과 멤버 함수

# Passing Arguments to Base Class Constructor

- 기본 클래스의 생성자로 인자 전달
  - 기본 클래스의 어떤 생성자를 호출할지 결정해 줄 수 있어야 함
  - 유도 클래스의 생성자에, 초기화 리스트를 활용해 사용자가 원하는 기본 클래스의 생성자 호출 가능

```
1 class Base {  
2 public:  
3     Base();  
4     Base(int);  
5     ...  
6 };  
7  
8 Derived::Derived(int x)  
9     : Base{x}, {...}  
10 {  
11     //derived constructor code  
12 }
```

*Base{x} 이므로 인자가 하나인 이 생성자가 호출됨.  
(만약 Base{} 였으면 위의 생성자가 호출되었을 것)*

*\*p.36의 예제에서 보인 바와 같이, 만일 초기화 리스트를  
활용해 어떤 기본 클래스 생성자를 호출할지 명시하지 않으면,  
기본 클래스에 대해서는 기본 생성자(인자가 없는 생성자)가  
호출되는 것이 default 동작.*

# Passing Arguments to Base Class Constructor

- 기본 클래스의 생성자로 인자 전달

```
1 class Base {  
2 private:  
3     int value;  
4 public:  
5     Base() : value{0} { cout << "Base no-args constructor" << endl; }  
6     Base(int x) : value{x} { cout << "Base (int) overloaded constructor" << endl; }  
7 };
```

- 기본 클래스의 생성자로 인자 전달

```
1 class Base {
2 private:
3     int value;
4 public:
5     Base() : value{0} { cout << "Base no-args constructor" << endl; }
6     Base(int x) : value{x} { cout << "Base (int) overloaded constructor" << endl; }
7 };
```

```
1 class Derived : public Base {
2 private:
3     int doubled_value;
4 public:
5     Derived() : Base{}, doubled_value {0} { cout << "Derived no-args constructor " << endl; }
6     Derived(int x) : Base{x}, doubled_value {x*2} { cout << "Derived (int) overloaded constructor" << endl; }
7 };
```

기본 클래스의 인자를 받지 않는 생성자를 호출하라

기본 클래스의 인자를 하나 받는 생성자를 호출하라



- 기본 클래스의 생성자로 인자 전달
  - P.36 결과와 비교!

```
1 int main() {  
2     Base b; //value: 0  
3     Base b{100}; //value: 100  
4     Derived d; //value: 0, doubled_value: 0  
5     Derived d {1000}; //value: 1000, doubled_value: 2000  
6 }
```

# Copy Constructor

---

- 복사 생성자의 상속

- 마찬가지로, 기본 클래스로부터 상속되지 않음
- (상속 전과 마찬가지로,) 컴파일러가 자동생성하지만, 필요한 경우 직접 구현 해야함
- 기본 클래스에서 구현한 복사 생성자 호출 가능

# Copy Constructor

- 유도 클래스의 복사 생성자
  - 기본 클래스의 복사 생성자를 직접 호출 가능
  - Slice 과정을 거침
  - (이동 생성자도 동일하게 동작)


```
1 Derived::Derived(const Derived &other)
2     : Base{other}, {Derived initialization list}
3 {
4     //code
5 }
```

*\*other은 유도 클래스이지만, slice를 통해 기본 클래스의 복사 생성자에 인수로 넘겨줄 수 있다. 유도 클래스 is a 기본 클래스 관계를 준수하기 때문!*

# Copy Constructor

- 유도 클래스의 복사 생성자, 예시

```
1 class Base{
2     int value;
3 public:
4     ...
5     Base(const Base &other) : value{other.value} {
6         cout << "Base Copy Constructor" << endl;
7     }
8 };
```

 Other에는 어떤 데이터가 들어있을까?

```
1 class Derived : public Base {
2     int doubled_value;
3 public:
4     ...
5     Derived(const Derived &other)
6         : Base{other}, doubled_value{other.doubled_value} {
7         cout << "Derived Copy Constructor" << endl;
8     }
9 };
```

- 복사 생성자의 구현 가이드

- 유도 클래스에서 사용자가 복사 생성자를 구현하지 않은 경우,
  - 컴파일러가 자동으로 생성하며, 기본 클래스의 복사 생성자를 호출
- 유도 클래스에서 사용자가 복사 생성자를 구현한 경우,
  - 명시하지 않으면 기본 클래스의 인자를 받지 않는 생성자를 호출
  - 기본 클래스를 위한 복사/이동 생성자를 명시적으로 호출해 주어야 함
- 따라서, 포인터형 멤버 변수를 가지고 있는 경우, 기본 클래스의 복사/이동 생성자를 호출하는 방법에 대해 반드시 숙지해 두어야 함
  - 유도 클래스 멤버 변수에 대한 깊은 복사 고려

# Inheritance

---

- 상속의 정의 : 기본 클래스를 기반으로 새 클래스 만드는 기능
- 유도 클래스 : 그렇게 만들어진 클래스를 유도 클래스라 하며, 기본 클래스의 모든 것이 포함되어 있음
- protected 멤버 : 상속에서 public은 그대로 public, private은 여전히 private이므로, 정보를 숨기되 상속 계층의 하위로 데이터를 상속하고 싶을 때는 protected를 사용
- 상속에서의 생성자와 소멸자 : 기본 생성-유도 생성 / 유도 소멸-기본 소멸 순서
- 기본 클래스 생성자와의 관계 : 유도 클래스 (복사)생성시에, 기본 클래스를 어떻게 (복사)생성할지를 초기화 리스트를 사용해 명시해 주어야 함
- 상속과 멤버 함수

- 기본 클래스의 멤버 함수 사용
  - 유도 클래스는 기본 클래스의 멤버 함수를 직접 호출 가능
    - Public / protected인 멤버 함수의 경우에
  - 유도 클래스는 기본 클래스의 멤버 함수를 **오버라이드** 또는 **재정의** 가능
  - 다형성의 구현을 위해 중요한 기능
- 재정의로 인해 기본 클래스 함수가 가려질 경우,
  - Base::Member()와 같은 형식으로 호출 가능

# Using Member Function of Base Class

- 기본 클래스의 멤버 함수 사용

- Player 예제

```
1 class Entity {
2 protected:
3     int x, y;
4 public:
5     Entity(int x, int y)
6         : x{ x }, y{ y }
7     {}
8     void Talk()
9     {
10         cout << "Hello";
11     }
12     void ShowPosition()
13     {
14         cout << x << "," << y << endl;
15     }
16 };
```

기본 클래스의 Talk함수를  
유도 클래스에서 재정의함!

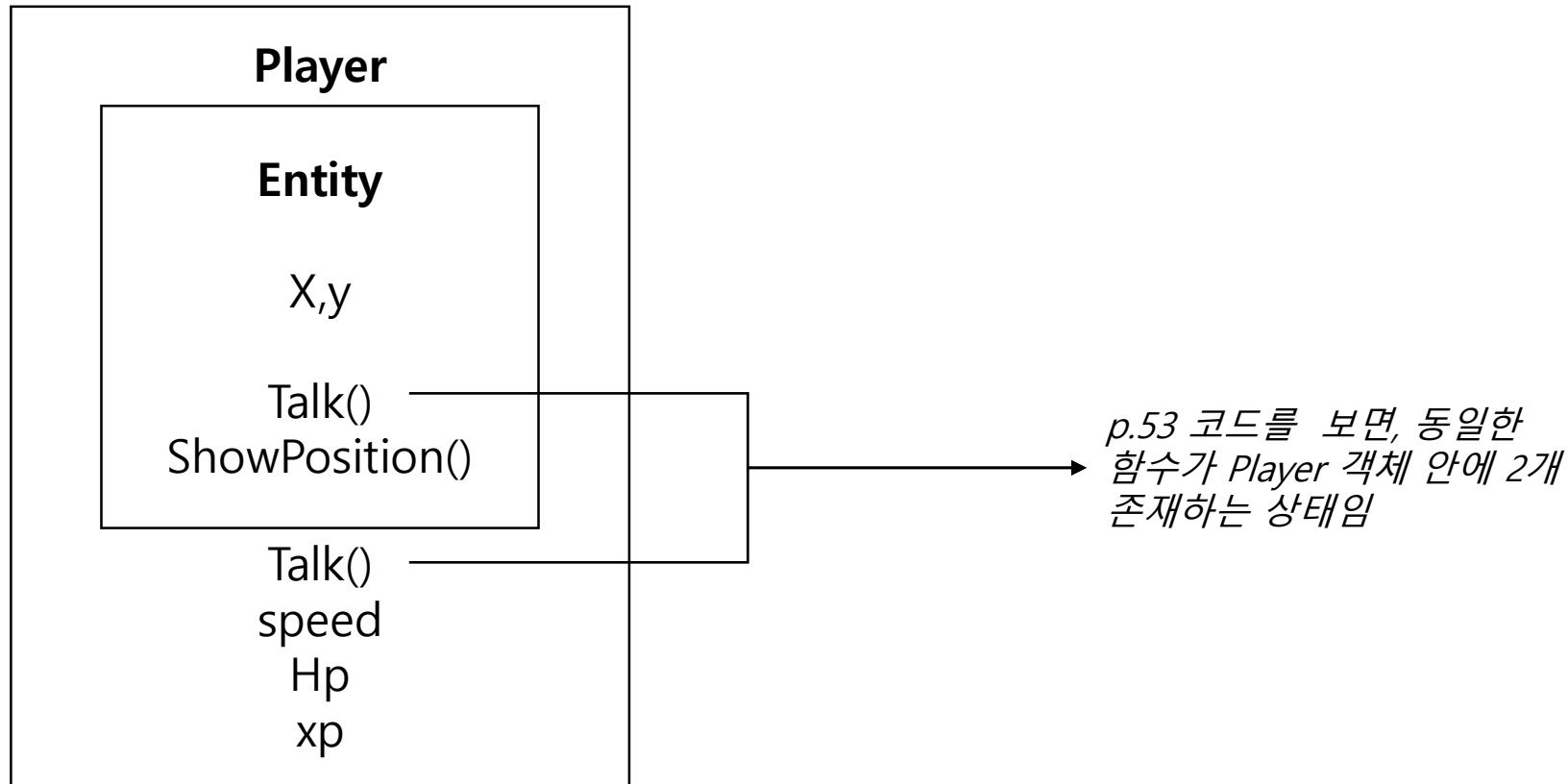
```
1 class Player : public Entity{
2 private:
3     int speed;
4     int hp;
5     int xp;
6 public:
7     Player(int x, int y, int speed)
8         : Entity{ x,y }, speed {speed}
9     {}
10    void Talk()
11    {
12        cout << "Hello. I'm Player";
13    }
14 };
```



# Using Member Function of Base Class

## ● 정적 바인딩

- 이전 그림대로라면, 유도 클래스와 기본 클래스에 같은 이름과 인자를 갖는 함수가 2개 존재
- 어떤 기준으로 호출할 함수를 결정할까? → 타입을 기준 호출!



# Using Member Function of Base Class

- 정적 바인딩과 그 한계

- 정적 바인딩은, 컴파일 시 어떤 함수가 호출될지를 결정하는 방식
- C++의 기본은 정적 바인딩

\*가장 아래의 경우, 동적 바인딩을 사용하여 기능의 확장 가능(다음 강의)

```
1 Entity e{1,1};
2 e.Talk(); //Entity::Talk call
3
4 Player p{1,1,2};
5 p.Talk(); //Player::Talk call
6
7 Entity *ePtr = new Player{1,1,2};
8 ePtr->Talk(); //??
```

*e는 Entity 타입이므로, e.Talk()는 Entity의 Talk 함수 호출*

*p는 Player 타입이므로, p.Talk()는 Player의 Talk 함수 호출*



*//1) 왼쪽이 가능한가?*

*//2) 가능한 경우 어떤 Talk 함수가 호출될 것인가?*

### ● "Is-A" 관계

#### ■ 기본 클래스 객체를 사용하는 곳에는 항상 유도 클래스 객체를 사용 가능

- 유도 클래스에 기본 클래스의 멤버가 모두 포함되기 때문 (superset)
- Player 객체는 두 가지 타입을 모두 가진 상태가 됨!

```
1 void TalkSomething(Entity e)
2 {
3     e.Talk();
4 }
5
6 void ShowSomething(Entity e)
7 {
8     e.ShowPosition();
9 }
10
11 int main()
12 {
13     Player p{ 0,0,1 };
14     TalkSomething(p);
15     ShowSomething(p);
16 }
```

```
1 int main()
2 {
3     Entity p = Player{1,1,1};
4     Entity* ePtr = new Player{ 1,1,1 };
5     Entity& pRef = p;
6     pRef.ShowPosition();
7
8     Player* pPtr = new Entity{ 1,1,1 };
9 }
```



\*Entity 타입이, Player 타입을 가리켜도 오류 없음  
\*반대는 안될까?

# Using Member Function of Base Class

- 기본 클래스의 멤버 함수 사용

- Player 예제

```
1 class Entity {
2 protected:
3     int x, y;
4 public:
5     Entity(int x, int y)
6         : x{ x }, y{ y }
7     {}
8     void Talk()
9     {
10         cout << "Hello";
11     }
12     void ShowPosition()
13     {
14         cout << x << "," << y << endl;
15     }
16 };
```

"가려진" 기본 클래스  
멤버를 유도 클래스에서  
접근하고 싶다면 오른쪽과  
같이 호출 가능

```
1 class Player : public Entity{
2 private:
3     int speed;
4     int hp;
5     int xp;
6 public:
7     Player(int x, int y, int speed)
8         : Entity{ x,y }, speed {speed}
9     {}
10    void Talk()
11    {
12        Entity::Talk();
13        cout << "I'm Player";
14    }
15 };
```

# (Summary) Inheritance

---

- 상속의 정의 : 기본 클래스를 기반으로 새 클래스 만드는 기능
- 유도 클래스 : 그렇게 만들어진 클래스를 유도 클래스라 하며, 기본 클래스의 모든 것이 포함되어 있음
- protected 멤버 : 상속에서 public은 그대로 public, private은 여전히 private이므로, 정보를 숨기되 상속 계층의 하위로 데이터를 상속하고 싶을 때는 protected를 사용
- 상속에서의 생성자와 소멸자 : 기본 생성-유도 생성 / 유도 소멸-기본 소멸 순서
- 기본 클래스 생성자와의 관계 : 유도 클래스 (복사)생성시에, 기본 클래스를 어떻게 (복사)생성할지를 초기화 리스트를 사용해 명시해 주어야 함
- 상속과 멤버 함수 : 유도 클래스에서, 기본 클래스의 함수 재정의 가능. 호출은 정적 바인딩 기준으로 이루어짐. "Is-A" 관계 철학에 따라 기본 클래스가 사용되는 곳에는 유도 클래스 사용 가능