Chapter 10:
상속 활용하기

Mijin An
meeeeejin@gmail.com

VLDB Lab.

클래스 확장하기

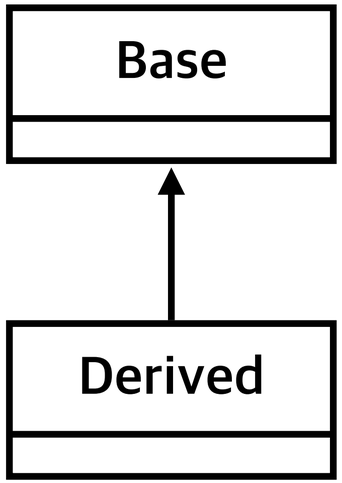
- 용어 정의
  - **원본** 클래스 = **부모** 클래스 = **베이스** 클래스 = **슈퍼** 클래스
  - **자식** 클래스 = **파생** 클래스 = **서브** 클래스

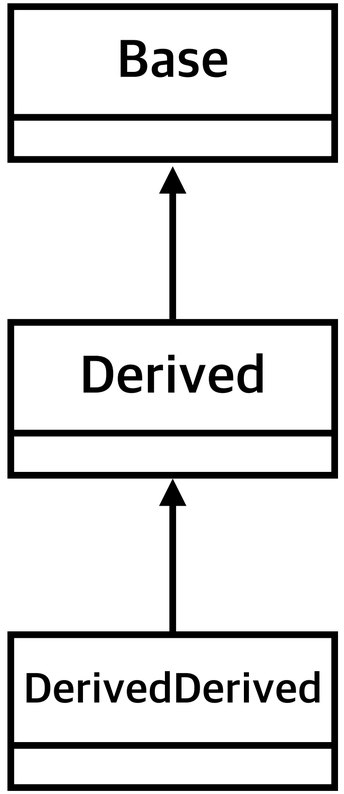
- C++에서 클래스를 정의할 때,
  컴파일러에 기존 클래스를 **상속 (inherit), 파생 (derive), 확장 (extend)** 한다고 선언

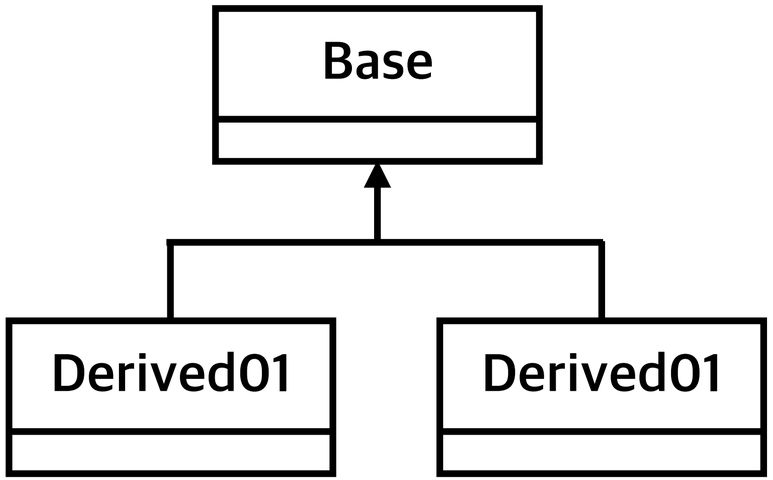
```
class Base
{
    public:
        void someMethod();
    protected:
        int mProtectedInt;
    private:
        int mPrivateInt;
};
```


```
class Derived : public Base
{
    public:
        void someOtherMethod();
}
```

상속의 다양한 유형


Base
Derived


Base
Derived
DerivedDerived


Base
Derived01
Derived01

클라이언트 입장에서 본 상속

- 상속은 반드시 **한 방향**으로만 진행
  - Base 타입 객체는 Derived 객체의 메서드나 데이터 멤버 사용 불가능

```
Base myBase;
myBase.someOtherMethod(); // Compile error!!
```

- 객체를 선언한 클래스의 객체뿐만 아니라 파생 클래스 객체도 가리킬 수 있음

```
// Derived 객체를 생성해서 Base 포인터에 저장
Base* base = new Derived();

base->someOtherMethod(); // Compile error!!
```

파생 클래스 입장에서 본 상속

- 파생 클래스는 **부모 클래스에 선언된** `public` **및** `protected` **메서드나 데이터 멤버**를 마치 자신이 정의한 것처럼 사용 가능

```
void Derived::someOtherMethod() {
    cout << "I can access base class data member " << mProtectedInt << endl;
}
```

- 파생 클래스에서의 접근 지정자
  - `Protected`: 파생 클래스에서 **접근 가능**
  - `Private`: 파생 클래스에서 **접근 불가능**
    - **나중에 정의될 파생 클래스에서 현재 클래스에 접근하는 수준 제어할 때 활용**
    - **기본적으로 데이터 멤버를 모두** `private`**으로 선언하는 것이 바람직!**

```
void Derived::someOtherMethod() {
    cout << "I can access the private member " << mPrivateInt << endl; // Error!!
}
```

상속 방지

- 클래스를 정의할 때, `final` 키워드를 붙이면 다른 클래스가 이 클래스를 상속할 수 없음

```
class Base final
{
    // 어쩌고저쩌고
};
```

메서드 오버라이딩: 베이스 클래스에서의 정의

- 클래스를 **상속**하는 주된 이유:
  - 기능 추가
  - 기능 변경

- 베이스 클래스에 정의된 메서드의 동작 변경 → **메서드 오버라이딩** 이용
  - `virtual` 키워드로 선언된 메서드만 오버라이딩 가능

```
class Base
{
    public:
        virual void someMethod();
    protected:
        int mProtectedInt;
    private:
        int mPrivateInt;
};
```

메서드 오버라이딩: 파생 클래스에서의 활용

- `override` 키워드를 사용해 베이스 클래스의 메서드를 오버라이드 가능

```
class Derived : public Base
{
    public:
        virtual void someMethod() override;
        virtual void someOtherMethod();
};
```

- 메서드나 소멸자를 `virtual`로 지정하면, 모든 파생 클래스에서도 `virtual` 상태 **유지**

메서드 오버라이딩: 클라이언트 관점

- 포인터/레퍼런스는 파생 클래스까지 가리킬 수 있음
  - 💡 **베이스 클래스 타입의 포인터/레퍼런스가 실제로 파생 클래스 타입 객체를 가리킨다 해도, 베이스 클래스에 정의되지 않았으면 접근 불가능!**

```
Derived myDerived;
Base& ref = myDerived;
myDerived.someOtherMethod(); // 정상 작동
ref.someOtherMethod();       // Error!!
```

- 파생 클래스를 베이스 클래스로 **캐스팅/대입은 가능**하나, 그 순간 **파생 클래스 정보는 사라짐**
  - **슬라이싱 (Slicing)**

```
Derived myDerived;
Base assignedObject = myDerived; // Base 변수에 Derived 객체 대입
assignedObject.someMethod();     // Base 버전의 someMethod() 호출
```

override 키워드의 중요성 (1)

```
class Base
{
    public:
        virtual void someMethod(double d);
};

class Derived : public Base
{
    public:
        virtual void someMethod(double d);
};

Derived myDerived;
Base& ref = myDerived;
ref.someMethod(1.1); // Derived 버전의 someMethod() 호출
```

override 키워드의 중요성 (2)

```
class Base
{
    public:
        virtual void someMethod(double d);
};

class Derived : public Base
{
    public:
        virtual void someMethod(int i);
};

Derived myDerived;
Base& ref = myDerived;
ref.someMethod(1.1); // Base 버전의 someMethod() 호출
```

새로운 virtual 메서드 생성

💡 베이스 클래스의 메서드를 오버라이드할 때는 항상 override 키워드를 사용‼️

virtual 키워드의 중요성 (1)

```
class Base
{
    public:
        void go() { cout << "called on Base" << endl; }
};

class Derived : public Base
{
    public:
        void go() { cout << "called on Derived" << endl; }
};

Derived myDerived;
Base& ref = myDerived;
ref.go(); // Base 버전의 go() 호출
```

`virtual` 키워드의 중요성 (2)

- `virtual`로 선언하지 않은 메서드를 오버라이드하면, **베이스 클래스 정의를 숨겨버림**

- 컴파일러는 `virtual`로 선언되지 않은 메서드를 호출하는 부분을 **컴파일 타임에 결정된 타입의 코드로 교체**
  - **정적 바인딩 (Static binding), 이른 바인딩 (Early binding)**

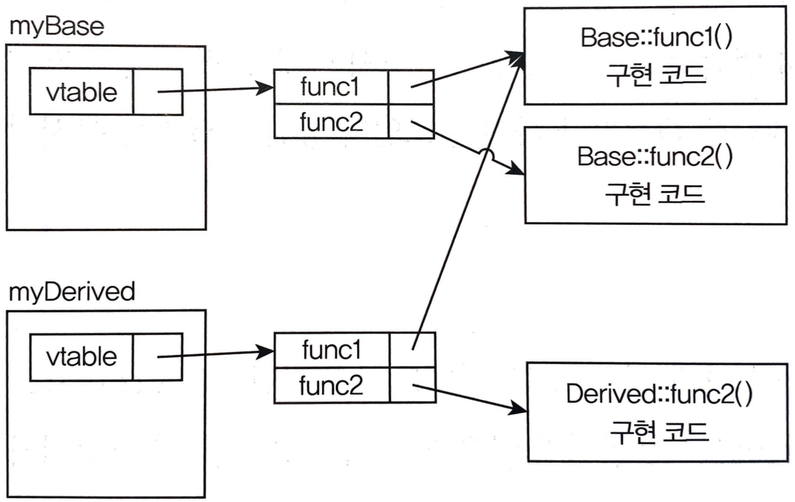
- 메서드를 `virtual`로 선언하면 **vtable**이라 부르는 **특수한 메모리 영역**을 활용해 가장 적합한 구현 코드 호출
  - **동적 바인딩 (Dynamic biding), 늦은 바인딩 (Late biding)**
    1. `virtual` 메서드가 하나 이상 정의된 클래스마다 vtable 하나씩 할당
    2. 이 클래스로 생성한 객체마다 vtable에 대한 포인터 가짐
    3. 객체에 대해 메서드 호출 → vtable을 보고 적합한 버전의 메서드 실행

virtual 키워드의 중요성 (3)

```cpp
class Base
{
    public:
        virtual void func1() {}
        virtual void func2() {}
        void nonVirtualFunc() {}
};

class Derived : public Base
{
    public:
        virtual void func2() override {}
        void nonVirtualFunc() {}
};

Base myBase;
Derived myDerived;
```



오버라이딩 방지

- 메서드를 정의할 때, `final` 키워드를 붙이면 다른 클래스가 이 메서드를 오버라이드할 수 없음

```
class Base
{
    public:
        virtual ~Base() = default;
        virtual void someMethod() final;
};

class Derived : public Base
{
    public:
        virtual void someMethod() override; // Compile error!!
};
```

코드 재사용을 위한 상속

- 파생 클래스에 **기능 추가**하기
  - e.g., 파생 클래스와 베이스 클래스 사이를 중계하는 인터페이스 추가
  - 부모 클래스의 명명 규칙 따르기

- 파생 클래스에서 **기존 기능 변경**하기
  - 오버라이드 후, 기능 변경

- 자세한 코드 (날씨 예측)는 책을 참고하세요 😇

부모를 공경하라

- 파생 클래스를 작성할 때, 반드시 **부모 클래스와 자식 클래스의 연동 방식**에 주의

- 부모 클래스의 **생성자**
  1. 베이스 클래스라면, 디폴트 생성자 실행
  2. static으로 선언하지 않은 데이터 멤버를 코드에 나타난 순서대로 생성
  3. 클래스 생성자의 본문을 실행

- 부모 클래스의 **소멸자**
  1. 현재 클래스의 소멸자 호출
  2. 현재 클래스의 데이터 멤버를 생성할 때와 반대 순서로 삭제
  3. 부모 클래스가 있다면, 부모의 소멸자 호출

부모를 공경하라: 예제

```
class Something
{
    public:
        Something() { cout << "2"; }
        virtual ~Something() { cout << "2"; }
};

class Base
{
    public:
        Base() { cout << "1"; }
        virtual ~Base() { cout << "1"; }
};

class Derived : public Base
{
    public:
        Derived() { cout << "3"; }
        virtual ~Derived() { cout << "3"; }
    private:
        Something mDataMember;
};
```

업캐스팅과 다운캐스팅

- **업캐스팅 (Upcasting)**: 베이스 클래스 타입으로 파생 클래스를 참조하는 것

```
Base &myBase = myDerived;
```

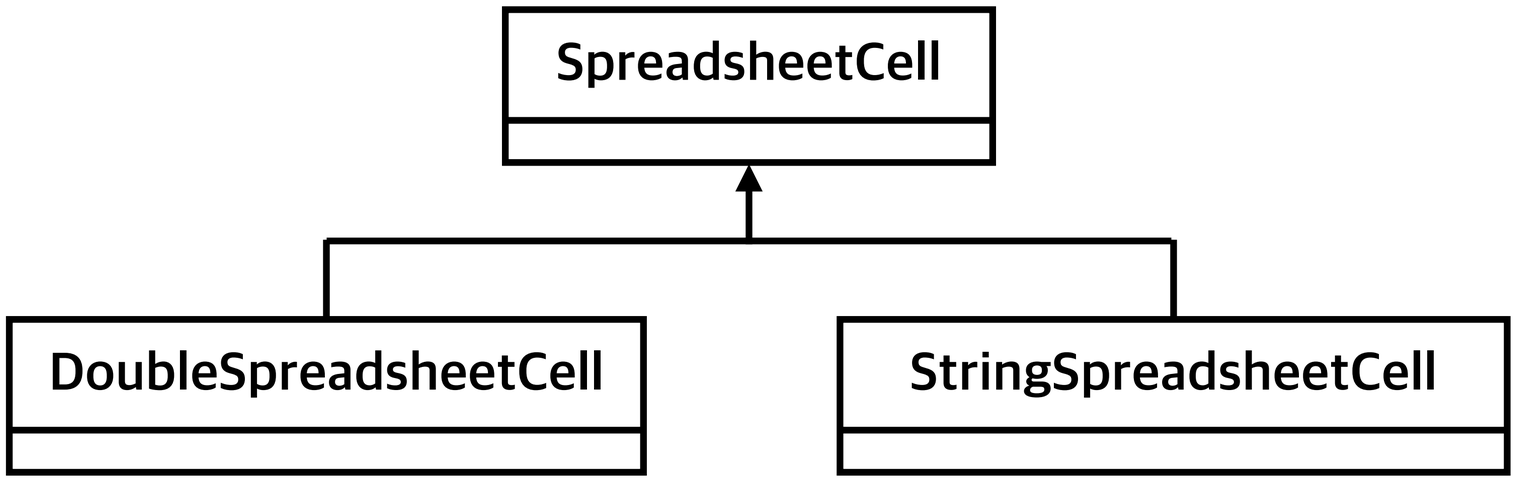
- **다운캐스팅 (Downcasting)**: 베이스 클래스를 파생 클래스로 캐스팅하는 것
  - 해당 객체가 반드시 파생 클래스에 속한다는 보장 없음
  - 디자인이 잘못된 것

```
void presumptuous(Base *base) {
    Derived* myDerived = static_cast<Derived*>(base);
    // myDerived로 Derived의 메서드에 접근하는 코드 작성
}
```

  - 통제 가능한 상황에서만 `dynamic_cast()`를 호출해 사용

```
void lessPresumptuous(Base *base) {
    Derived* myDerived = dynamic_cast<Derived*>(base);
    if (myDerived != nullptr) {
        // myDerived로 Derived의 메서드에 접근하는 코드 작성
    }
}
```

다형성을 위한 상속



- 같은 부모로부터 파생된 여러 타입을 마음껏 교체 가능
- 자세한 코드 (스프레드시트)는 책을 참고하세요 😇

순수 가상 메서드와 추상 베이스 클래스

- **순수 가상 메서드 (Pure virtual method)**
  - 클래스 정의 코드에서 명시적으로 정의하지 않는 메서드
  - 메서드 선언 뒤에 `=0` 을 붙임

- **추상 클래스 (Abstract class)**
  - 순수 가상 메서드가 최소한 하나라도 정의된 클래스

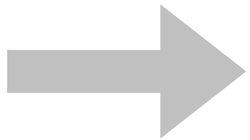
```
class SpreadsheetCell
{
    public:
        virtual ~SpreadsheetCell() = default;
        virtual void set(std::string_view inString) = 0;
        virtual std::string getString() const = 0;
};
```

다중 상속

```
class Dog
{
    public:
        virtual void bark() { cout << "Woof!" << endl; }
};

class Bird
{
    public:
        virtual void chirp() { cout << "Chirp!" << endl; }
};

class DogBird : public Dog, public Bird
{
    // 클래스 선언 코드
};
```



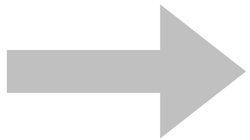
```
DogBird myConfusedAnimal;
myConfusedAnimal.bark(); // Woof!
myConfusedAnimal.chirp(); // Chirp!
```

다중 상속: 모호한 이름 (1)

```
class Dog
{
    public:
        virtual void bark() { cout << "Woof!" << endl; }
        virtual void eat() { cout << "The dog ate." << endl; }
};

class Bird
{
    public:
        virtual void chirp() { cout << "Chirp!" << endl; }
        virtual void eat() { cout << "The bird ate." << endl; }
};

class DogBird : public Dog, public Bird
{
    // 클래스 선언 코드
};
```



```
DogBird myConfusedAnimal;
myConfusedAnimal.eat(); // Error!!
```

다중 상속: 모호한 이름 (2)

- 해결 방법:
  1. `dynamic_cast()`로 객체를 **명시적으로 업캐스팅**
  2. **스코프 지정 연산자**로 원하는 버전을 구체적으로 지정

```
DogBird myConfusedAnimal;
dynamic_cast<Dog&>(myConfusedAnimal).eat(); // The dog ate.
myConfusedAnimal.Bird::eat();               // The bird ate.
```

- 파생 클래스에 이름이 같은 메서드가 있을 때도, 원하는 메서드 명확히 지정

```
class DogBird : public Dog, public Bird
{
    public:
        void eat() override;
};

void DogBird::eat() {
    Dog::eat(); // The dog ate.
}
```

OR

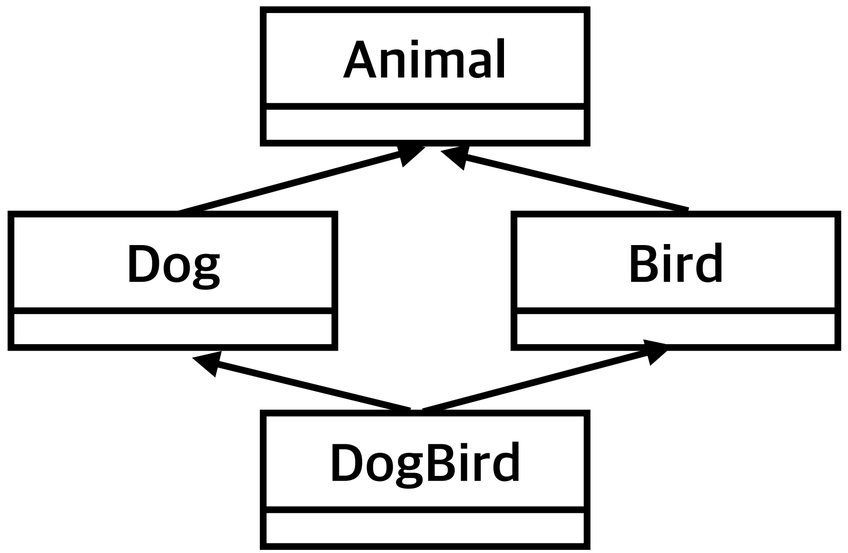
```
class DogBird : public Dog, public Bird
{
    public:
        using Dog::eat; // The dog ate.
};
```

다중 상속: 모호한 베이스 클래스

- 상속 관계가 이상하거나 클래스 계층이 정리되지 않았을 때 발생

```
class Dog {};
class Bird : public Dog {};
class DogBird : public Bird, public Dog {}; // Error!!
```

- 부모 클래스가 겹치는 경우 발생



오버라이드한 메서드의 속성 변경: Return Type

- 메서드 오버라이드 시, 베이스 클래스의 메서드 프로토타입과 똑같이 작성하는 것이 원칙

- **공변 리턴 타입 (Covariant return type)**
  - 베이스 클래스의 리턴 타입이 다른 클래스에 대한 포인터/레퍼런스이면,
    메서드를 오버라이드할 때 리턴 타입을 그 클래스의 파생 클래스에 대한 포인터/레퍼런스 타입으로 변경 가능
  - 베이스 클래스 – 파생 클래스가 **병렬 계층 (Parallel hierarchy)**일 때 유용
    - 두 계층이 따로 존재하지만, 어느 한쪽에 관련이 있을 때 유용

오버라이드한 메서드의 속성 변경: 매개변수

- `virtual` 메서드를 선언할 때,
  - 이름은 **부모 클래스와 똑같이** 쓰고
  - **매개변수만 다르게** 지정하면
  - 새로운 메서드가 정의됨

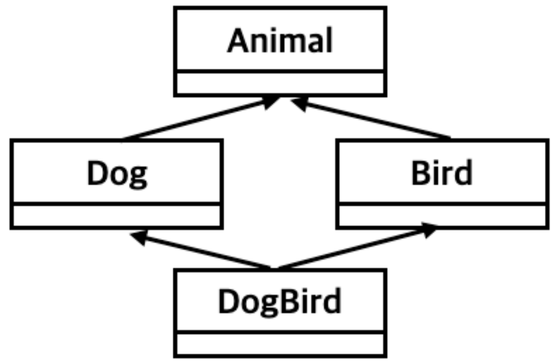
실행 시간 타입 정보

- C++은 컴파일 시간에 결정하는 것이 많음
  - **실행 시간에 객체를 들여다보는 기능 → RTTI (Run-Time Type Information)**
    - e.g., `dynamic_cast()`

- RTTI에서 제공하는 `typeid` 연산자:
  - **실행 시간에 객체의 타입 정보 조회** 가능
  - 기능 구현 시엔 사용하지 않음
    - 객체의 타입에 따라 다르게 실행되는 코드는 `virtual` 메서드로 구현
  - **로깅 및 디버깅 용도**로 활용

가상 베이스 클래스

- **동일한 클래스를 상속하는 부모 클래스를 여러 개 상속하면 모호함 발생**
  - 이를 해결하기 위해 등장




```cpp
class Animal
{
    public:
        virtual void eat() = 0;
        virtual void sleep() { cout << "zzzz.." << endl; }
};

class Dog : public virtual Animal
{
    public:
        virtual void bark() { cout << "Woof!" << endl; }
        virtual void eat() override { cout << "The dog ate." << endl; }
};

class Bird : public virtual Animal
{
    public:
        virtual void chirp() { cout << "Chirp!" << endl; }
        virtual void eat() override { cout << "The bird ate." << endl; }
};

class DogBird : public Dog, public Bird
{
    public:
        virtual void eat() override { Dog::eat(); }
};
```

```cpp
int main() {
    DogBird myConfusedAnimal;
    myConfusedAnimal.sleep();
    return 0;
}
```

Reference

[1] Marc Gregoire, “Professional C++, 4th Edition”, Wiley, 2018