Polymorphism (II)

Computer Programming for Engineers (DSAF003-42)

Instructor:

Youngjoong Ko (nlp.skku.edu)

This Week

- More about Virtual Functions
  - C++11 keywords: override, final
  - Pure Virtual Function
  - Abstract Base Class
  - Extended type compatibility
  - Downcasting and upcasting

MORE ABOUT VIRTUAL FUNCTIONS

C++11 override keyword

그냥 재정의함을 명확히 함.

- **override** clarifies if a function is overridden

```
class Sale
{
  public:
    …
    virtual double bill() const;
    …
};

class DiscountSale : public Sale
{
  public:
    …
    double bill() const override;
    …
};
```

Makes it explicit that this function overrides **bill()** in the Sale class

C++11 `final` keyword

- C++11 includes the `final` keyword
  - to prevent a function from being overridden. 함수가 override 되지 않게 함.
  - Useful if a function is overridden but don't want a derived classes to override it again. 더이상 override 안될 때 use.

```
class Sale
{
  public:
    virtual double bill() const final; // cannot override
};

class DiscountSale : public Sale
{
  public:
    double bill() const; // results in compiler error
};
```

- override and final keywords (4_pet_override_final.cpp)
  - Error cases
    - override keyword without overriding
    - final keyword with latter overriding

final 을 만나지 않 음!

```
class Dog : public Pet
{
  public:
    string breed;
    void print() const override;
};

class MyDog : public Dog
{
  public:
    string address;
    void print() const override;
};
```

[Error Cases – Where are they?]

```
class Dog : public Pet
{
  public:
    string breed;
    void print() const override final;
};

class MyDog : public Dog
{
  public:
    string address;
    void print() const override;
    void printAddr() const override;
};
```

Virtual Functions: Why Not All?

- One major disadvantage: overhead! 저장공간 개많이씀
  - Uses more storage (typically, by a size of a single pointer)
    - Internally, an additional pointer to **VTABLE** (virtual function table) is stored implicitly. 이 대한 추가 포인터 저장
    - **VTABLE** stores the data for virtual functions. 이 대한 data 저장
  - Late binding is "on the fly", so programs run slower 느려짐

- So if virtual functions not needed, should not be used.

Pure Virtual Functions

순수 가상함수 ; 반드시 재정의해야!

- Base class might not have "meaningful" definition for some of it's members!
  - It's purpose solely for others to derive from

- Recall class Figure
  - All figures are objects of derived classes: Rectangles, circles, triangles, etc.
  - Class Figure has no idea how to draw! draw가 뭔지?
  - Make it a pure virtual function by adding **"=0"**:

```
virtual void draw() = 0; // = 0 indicates pure virtual
```

Abstract Base Classes

추상 기본 Class

- Pure virtual functions require no definition
  - Forces all derived classes to define "their own" version

모든 DC가 자기만의 version을 정의하도록.

- Abstract base class (often, interface in other languages)
  - Classes with one or more pure virtual functions
  - No objects can ever be created from it.

어떤 객체도 생성 불가.

    - Since it doesn't have complete "definitions" of all it's members!
  - If derived class fails to define all pure's:

↳ 정의가 완전하지 않아.

    - It's an abstract base class too

- Pure virtual functions and abstract types (5_pet_pure_virtual.cpp)
  - Erroneous cases
    - Virtual functions without implementations in base classes
    - Creating objects of abstract types

추상 class는 객체 선언 X

```
class Pet
{
  public:
    string name;
    virtual void print() const = 0;
};
class Dog : public Pet
{
  public:
    string breed;
    void print() const override final;
};
void Dog::print() const
{
  Pet::print();
  cout << "breed: " << breed << endl;
}
```

안 돼~

[Error Cases – Where are they?]

```
int main()
{
  Dog dog;
  //Pet pet;

  dog.name = "Tiny";
  dog.breed = "Great Dane";

  dog.print();

  return 0;
}
```

private라 돼니까 접근 불가능.

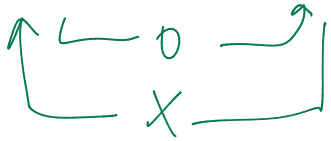
EXTENDED TYPE COMPATIBILITY AND SLICING PROBLEM

Extended Type Compatibility

- Given: Derived is derived class of Base

파생 객체는 기본 객체에 할당 가능

  - Derived objects can be assigned to objects of type Base (Derived → Base)
  - But, NOT the other way (i.e., Derived ← Base)!
    - We do not know how to assign the members Derived from Base.
  - Consider previous example:
    - A DiscountSale "is a" Sale, but reverse not true



Extended Type Compatibility Example

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

- Notice member variables name and breed are public!

Using Classes Pet and Dog

```
Dog vdog;
Pet vpet;

vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpet = vdog;
```

- dog "is a" pet:
  - These are allowable.
- Can assign values to parent-types, but not reverse
  - A pet "is not a" dog (not necessarily).

Slicing Problem

vpet = vdog 가 됐으니까 dog의 있는 breed 값 소실.

- Notice value assigned to vpet "loses" it's breed field!
  - Called **slicing problem**
    - `cout << vpet.breed;  // produces ERROR msg!`

- However, it might seem appropriate.
  - Dog was moved to `Pet` variable, so it should be treated like a `Pet`.
    - And therefore it does not have "dog" properties

dog 특성 소실.

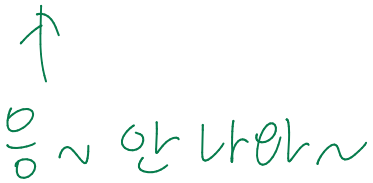
- Derived → Base, slicing problem example (6_pet_slicing_problem.cpp)
  - Erroneous cases
    - Base → Derived error
    - See the slicing problem in demo


```
class Pet
{
  public:
    string name;
    virtual void print() const;
    Pet() {}
    Pet(const Pet& pet) {
      cout << "in copy ctor(Pet)" << endl;
      name = pet.name;
    }
};
```

```
int main()
{
  ...
  Pet pet;
  pet = dog;
  //Pet pet = dog;

  // Following line is illegal
  //cout << pet.breed << endl;
  pet.print();

  return 0;
}
```



Slicing Problem Example

```
Pet* ppet;
Dog* pdog;
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Great Dane";
ppet = pdog;

// Cannot access breed field of object pointed to by ppet:
cout << ppet->breed;   // ILLEGAL!
```

- In C++, slicing problem is a nuisance

- Fix to slicing problem in C++
  - We'd like to refer to it's breed even if it's been treated as a Pet.
  - **We can do so with pointers to dynamic variables**

Slicing Problem Example

- Must use virtual member function:

```
ppet->print();
```

  - Calls print member function in Dog class!
    - Because it's virtual.

  - C++ "waits" to see what object pointer ppet is actually pointing to before "binding" call

바인딩 호출 전에 실제로 가리키는 객체 포인터 ppet이 뭔지 기다린다.

BC 포인터에 DC 객체를 담고 func을 부를 수 있.

- Resolving slicing problem using pointers (7_pet_slicing_pointer.cpp)


```
int main()
{
  Dog* dog;
  Pet* pet;
  dog = new Dog;

  dog->name = "Tiny";
  dog->breed = "Great Dane";
  pet = dog;
  // Following line is still illegal
  //cout << pet->breed << endl;
  pet->print();
  return 0;
}
```

애가 실행될 Dog니까~

Virtual Destructors

- Recall:
  - destructors needed to de-allocate dynamically allocated data
- Consider:

```
Base *pBase = new Derived;
…
delete pBase;  //Which destructor is called when ~Derived() is virtual
```

  - Would call base class destructor even though pointing to Derived class object!
  - Making destructor virtual fixes this!

- Good policy for all destructors to be virtual

Inner Workings of Virtual Functions

- Virtual function table (VTABLE)
  - Compiler (implicitly and automatically) creates it
  - Has pointers for each virtual member function
  - Points to location of correct code for that function
  - Do not try to hack VTABLE, because its implementation may differ

- However, don’t need to know how to use it!
  - Principle of information hiding