Polymorphism (I)

Computer Programming for Engineers (DSAF003-42)

Instructor:

Youngjoong Ko (nlp.skku.edu)

This Week

- Virtual Function Basics
  - Late binding
  - Implementing virtual functions
  - When to use a virtual function
  - Pure virtual functions and abstract classes

WARMING UP

Polymorphism?

- Definition
  - **The same entity (function or object) behaves differently in different scenarios**

- **Overloading**
  - **same function name, different data type or number of parameter**
    - Function Overloading:
      - func(int inumber) {cout<< ”print integer” << inumber << endl;};
      - func(float fnumber) { cout<< ”print float” << fnumber << endl;};
    - Operator overloading:
      - “+” operator in C++ can perform two specific functions at two different scenarios
      - For numbers, it performs addition.
      - For strings, it performs concatenation.

Polymorphism?

- **Overriding**

BC의 func. 를 DC에서 재정의

  - **New definition to base class function in the derived class**

<=> redefined.
아닌 가상함수 X.

    - Base class:
      - class Vehicle
        { public:
          void move(){cout << “move” <<endl;}}
    - Derived classes:
      - class Car: public Vehicle
        { public:
          void move(){cout<<“move with 4 wheels” <<endl;}}
      - class Bicycle: public Vehicle
        { public:
          void move(){cout<<“move with 2 wheels” <<endl;}}

Virtual Usage Examples (1_vehicle.cpp, 2_vehicle_virtual.cpp)


```
class Vehicle
{
  public:
    void move() {cout << “Vehicle: move”
                 << endl;}
};

class Car : public Vehicle
{
  public:
    void move() {cout << “Car: move with 4
                 wheels” << endl;}
};

class Bicycle : public Vehicle
{
  public:
    void move() {cout << “Bicycle: move with 2
                 wheels” << endl;}
};
```

```
int main()
{
    Vehicle vehicle;
    vehicle.move();

    Car car;
    car.move();

    Bicycle bicycle;
    bicycle.move();

    return 0;
}
```

Late Binding, Abstract Base Class

- **Late Binding (dynamic binding, Run time Polymorphism)**
  - Determine to implement a procedure in run time
  - With virtual function

- **Pure Virtual Function**
  - Virtual function in C++ for which we need not to write any function definition and only we have to declare it.
  - Ex) virtual void draw()=0;

- **Abstract Class**
  - Class that is designed to be specifically used as a base class
  - Class with at least one pure virtual function

VIRTUAL FUNCTION BASICS

Virtual Function Basics

- Polymorphism
  - Associating many meanings to one function
  - Virtual functions provide this capability
  - Fundamental principle of object-oriented programming!

- Virtual
  - Exist "essence (function header)" but does not exist definition of the function

- Virtual Function
  - Can be "used" before it's "defined"

Figures Example

- Classes for several kinds of figures
  - Each figure is an object of different class
    - Rectangle data: height, width, center point
    - Circle data: center point, radius

모든 수치 부분 클래스에서 파생.

- All figures are derived from one parent-class: Figure
  - Require same function: draw()
  - Each class needs different draw function 각 class마다 이름 draw 필요.
  - Can be called "draw" in each class:

```
Rectangle r; Circle c;
r.draw();  //Calls Rectangle class's draw
c.draw();  //Calls Circle class's draw
```

Problems in Figures Example

- Class Figure contains functions that apply to "all" figures
- Problem description
  - Consider a function `center()` that moves a figure to the center of screen
  - Example pseudo code of `center()` :

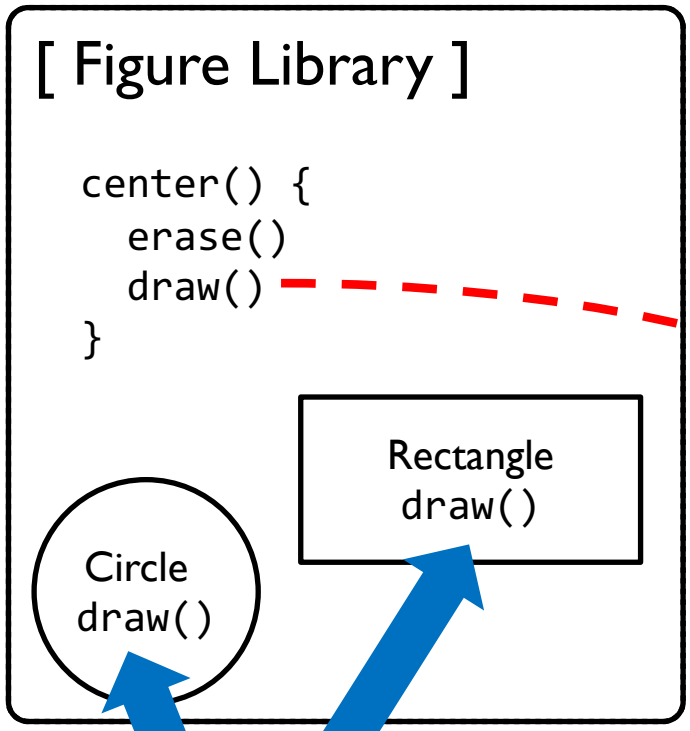
```
Figure::center() {
   eraseFigure()            // firstly, erase the figure
   draw(screenCenter)       // and then re-draw the figure
}
```

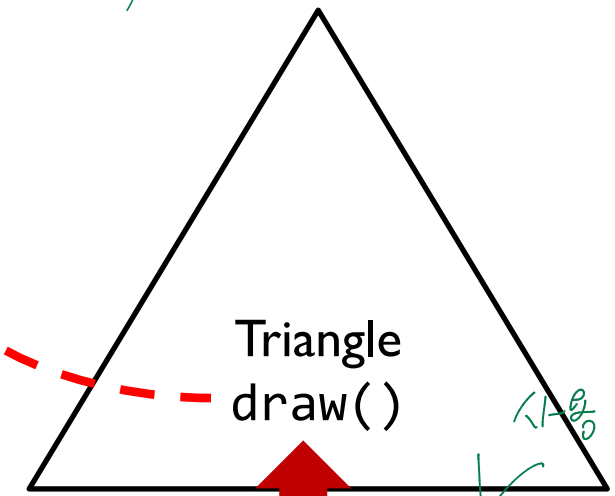
  - So, Figure::center() would use function draw() to re-draw.
  - Complications!
    - Which draw() function? From which class?

Problems in Figures Example

- Consider a new kind of figure comes along:
  - Triangle class derived from Figure class
- Function `center()` inherited from Figure
  - Will it work for triangles?
  - It uses `draw()`, which is different for each figure!
  - It will use `Figure::draw()` → won't work for triangles
- Want inherited function `center()` to use function `Triangle::draw()` NOT function `Figure::draw()`
  - But class `Triangle` wasn't even WRITTEN when `Figure::center()` was! Doesn't know "triangles"!

Graphical Exaplanation





2. Adding a new derived class.
`Figure` library DOES NOT know class `triangle`.

1. Compiled with two derived classes, and now in use.

Problem1 : `Figure::center` is confused about which `draw()` to call.

Problem 2: The `draw` function in `Triangle` is not in the in-use library, and we need to connect `draw` functions (binding).

Answer in Figures Example: Virtual!

- Tells compiler:
  - "Don't know how function is implemented"
  - "Wait until used in program"
  - "Then get implementation from object instance"

- Virtual functions are the answer
  - Called late binding or dynamic binding
  - Virtual functions implement late binding

Virtual Functions: Another Example

- Record-keeping program for automotive parts store
  - Track sales, but don't know all sales yet
  - First only regular retail sales (Regular prices)
  - Later: Discount sales (Discounted prices), mail-order, etc.
  - Program must:
    - Compute daily gross sales, Calculate largest/smallest sales of day
    - Perhaps average sale for day
- All come from individual bills
  - But many functions for computing bills will be **added "later"**!
    - When different types of sales added!
  - **So function for "computing a bill" will be virtual!**

Class Sale: Definition

- Display 15.1 Interface for the Base Class Sale

```
class Sale {
public:
    Sale();
    Sale(double thePrice);
    double getPrice() const;
    virtual double bill() const;
    double savings(const Sale& other) const;
private:
    double price;
};
```

나중에 Sale 타입의 bill 호출 가능.

- Note that "virtual" in declaration of member function `bill`
  - Later, derived classes of `Sale` can define **THEIR** versions of function `bill`
  - Other member functions of `Sale` will use version of derived class!
  - They won't automatically use `Class Sale's` version!

Class Sale: Member Functions

- `savings()` and `operator <`
  - Notice BOTH use member function bill() (i.e., virtual function)

```
double Sale::savings(const Sale& other) const
{
   return (bill() – other.bill());
}

bool operator<( const Sale& first, const Sale& second)
{
   return (first.bill() < second.bill());
}
```

  - We can overload operators, which is called ***operation overloading***.
    - This topic will be covered in detail later.

Derived Class DiscountSale Defined

- Display 15.3 Interface for the Derived Class DiscountSale

```
class DiscountSale : public Sale
{
public:
    DiscountSale();
    DiscountSale(double thePrice, double theDiscount);
    double getDiscount() const;
    void setDiscount(double newDiscount);
    virtual double bill() const;
private:
    double discount;
};
```

bill이 B.C에서 virtual 선언돼서

Since bill was declared virtual in the base class,
it is automatically virtual in the derived class `DiscountSale`
even without "virtual" keyword. 자동으로 가상이 됨.
But, it is recommended to add "virtual" 그래도 적어도 좋다.
to explicitly indicate it's virtual for readability.

DiscountSale's Implementation of bill()

```
double DiscountSale::bill() const
{
    double fraction = discount/100;
    return (1 – fraction)*getPrice();
}
```

- Late binding
  - In `savings()` and "<", `bill()` function in the derived class `DiscountSale` is called when the object is the one from `DiscountSale` class.
  - Because bill() is virtual and the function is dynamically bound.

Virtual: Wow!

DC 생성 이전에 정의된 BC.

- Recall class Sale written before derived class DiscountSale
  - Members savings and "<" are compiled before we even had ideas of a DiscountSale class

즉 로직 형성 전에 컴파일 됨.

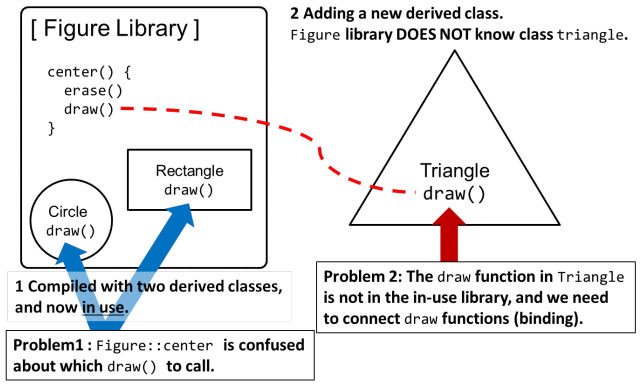
- Yet in a call like:

```
DiscountSale d1, d2;
d1.savings(d2);
```

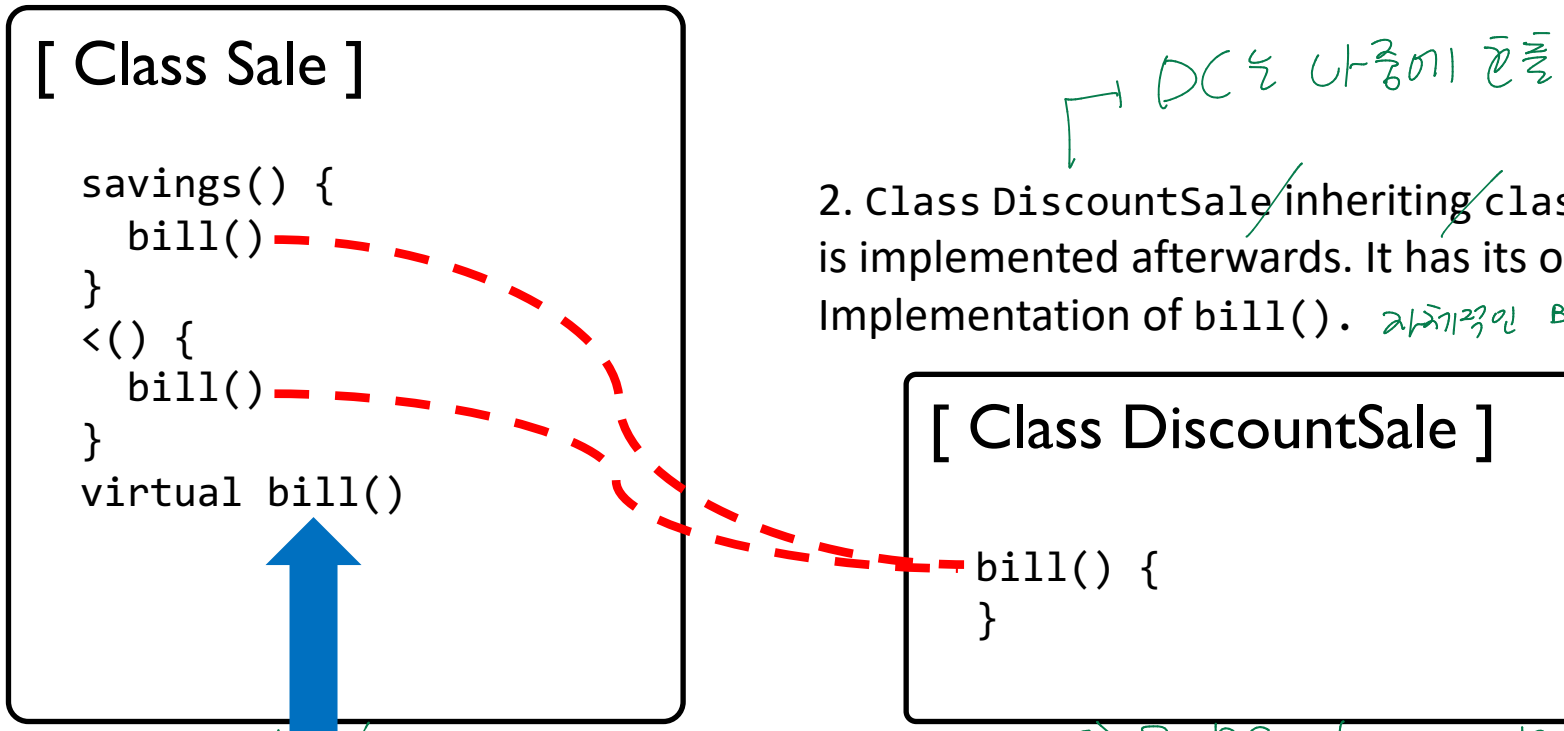
↳ 어떻? virtual로 선언해서.

BC::saving()이 DC::bill() 호출함

  - Call in `savings()` to function `bill()` knows to use definition of `bill()` from `DiscountSale` class.
  - Remind the dotted line in the previous figure.



Graphical Explanation 2



2. Class DiscountSale inheriting class Sale is implemented afterwards. It has its own Implementation of bill().

1. Class Sale has savings() and <() and they use bill().

Situation 1 : bill() is virtual, so it is not bound in compile time.

Situation 2: . But, when calling savings and "<", bill() in this derived class is called with late binding.

Virtual: How?

- To write C++ programs:
  - Assume it happens by "magic"!

- But explanation involves late binding
  - Virtual functions implement late binding.
  - Tells compiler to "wait" until function is used in program.
  - Decide which definition to use based on calling object
  - **Very important OOP principle!**

Overriding

B에서 가상 function 정의 변경

- Virtual function definition changed in a derived class
  - We say it's been "overidden"
  - Similar to *redefined*

- So:
  - Virtual functions changed: overridden
  - Non-virtual functions changed: redefined

Virtual Usages (3_pet.cpp)


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

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

void Pet::print() const
{
  cout << "name: " << name << endl;
}
```

```
void Dog::print() const
{
   cout << "breed: " << breed << endl;
}

int main()
{
  Dog dog;

  dog.name = "Tiny";
  dog.breed = "Great Dane";
  dog.print();

  return 0;
}
```