Polymorphism

Computer Programming for Engineers
(DASF003-41)

Instructor:

Sungjae Hwang (jason.sungjae.hwang@gmail.com)

Today

Virtual Function Basics

- Late binding
- Implementing virtual functions
- When to use a virtual function
- Abstract classes and pure virtual functions

Pointers and Virtual Functions

- Extended type compatibility
- Downcasting and upcasting
- C++ "under the hood" with virtual functions

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

- Existing in "essence" though not in fact

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: height, width, center point, draw()
  - Circle: center point, radius, draw()

All derived from one parent-class: Figure

- Require function: draw()
- Each class needs different draw function
- 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
```

- Nothing new here yet…

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?

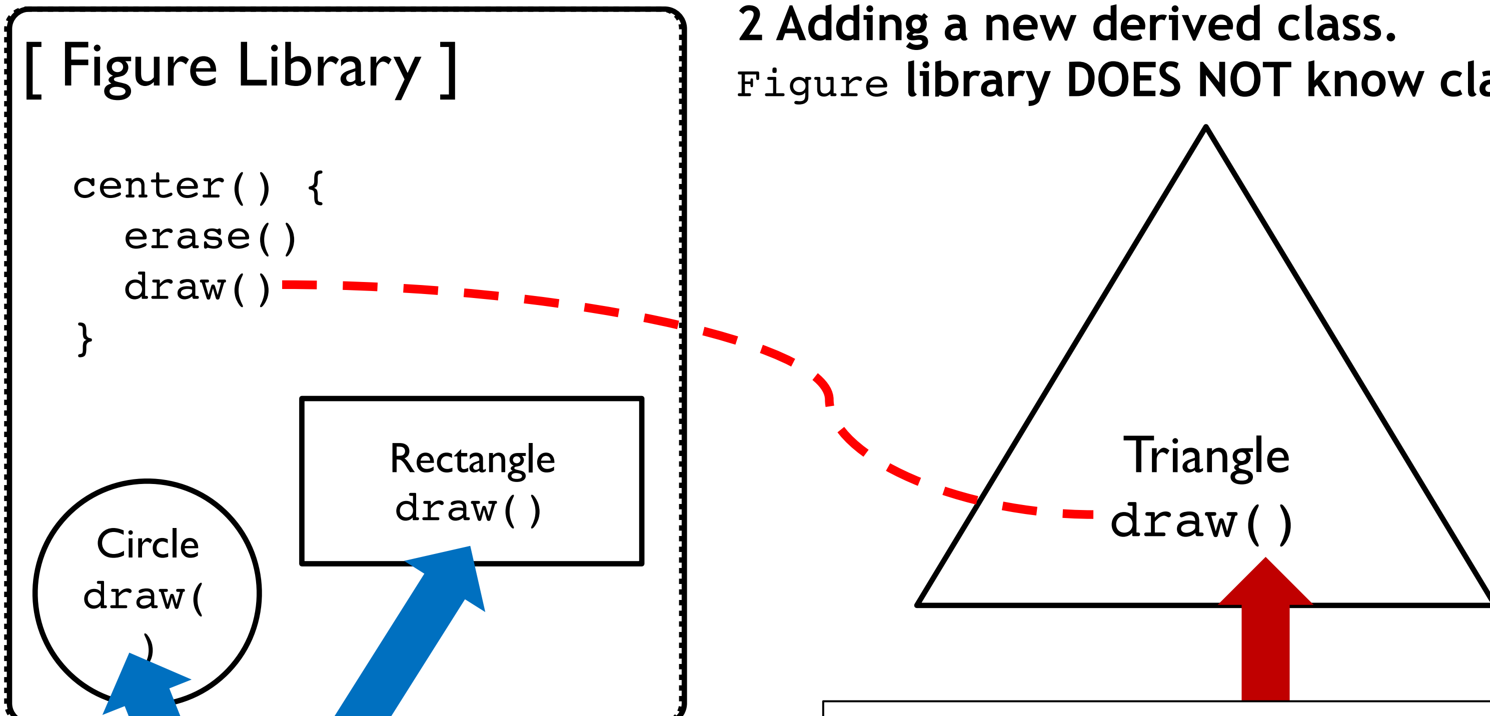
Problems in Figures Example

- **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 Explanation

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



1 Compiled with two derived classes, and now <u>in use</u>.

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

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

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 an 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;
};
```

- **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()`

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

  - Notice it use member function bill() (i.e., virtual function)
  - Which bill function will be invoked for “other.bill()”?

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;
};
```

Since bill was declared virtual in the base class, it is <u>automatically virtual</u> in the derived class `DiscountSale`
even without "`virtual`" keyword.
But, it is recommended to add "`virtual`" 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();
}
```

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

Late binding

- In `savings()`, `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.

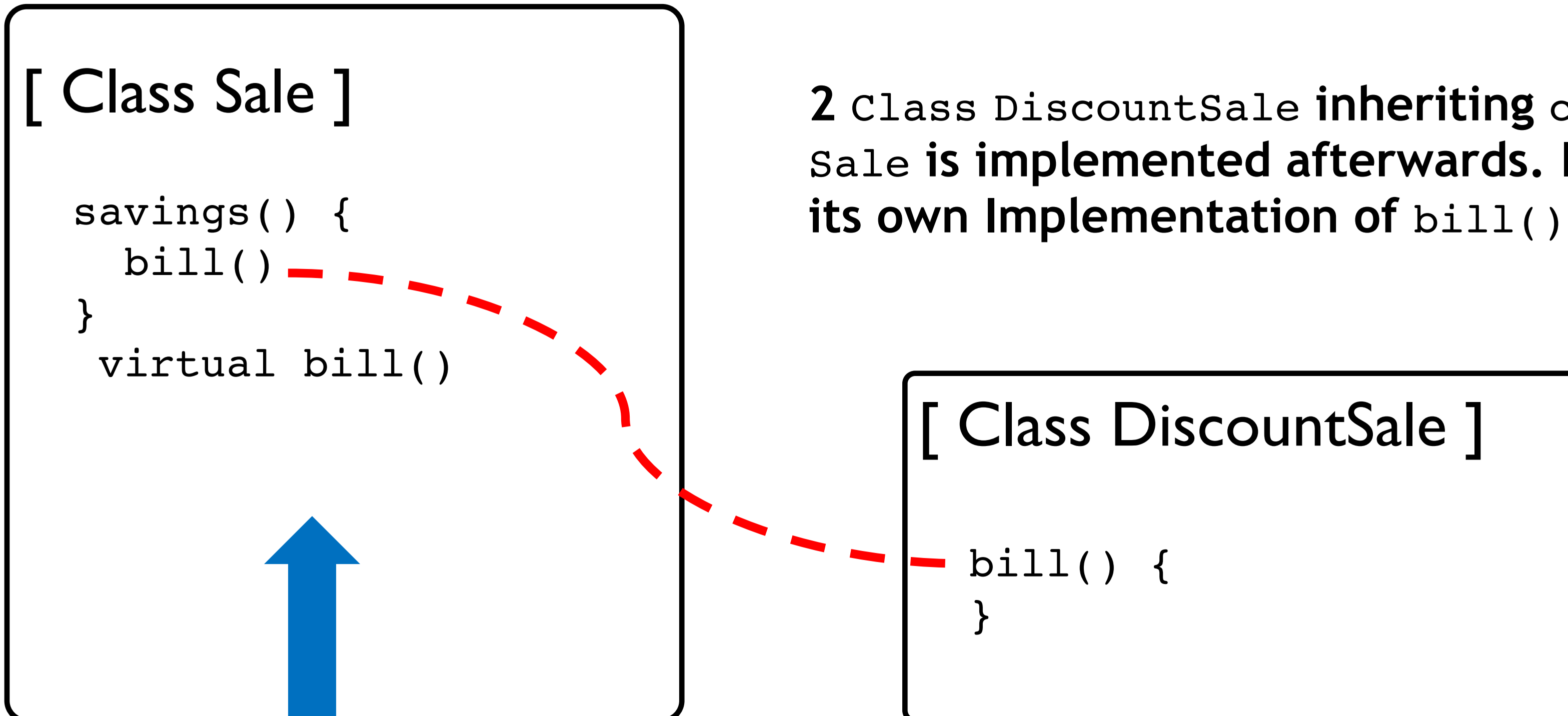
DiscountSale's Implementation of bill()

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

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

- **Late binding**
  - In `savings()`, `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.

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 it uses** `bill()`**.**

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

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

Virtual Function

DEMO

```
#include <iostream>
using namespace std;

class A{
  public:
    virtual void printv(){
      cout << "A class printv func is called" << endl;
    }

    void print(){
      cout << "A class print func is called" << endl;
    }
};

class B : public A{
  public:
    virtual void printv(){
      cout << "B class printv func is called" << endl;
    }

    void print(){
      cout << "B class print func is called" << endl;
    }
};

int main(){
  A* ptrA;
  B objB;
  ptrA = &objB;

  ptrA->print();
  ptrA->printv();
  return 0;
}
```

Polymorphism

```
#include <iostream>
using namespace std;

class sale{
  public:
  virtual void print(){
    cout << "Sale" << endl;
  }
};

class discountSale : public sale{
  public:
  virtual void print(){
    cout << "Discount Sale" << endl;
  }
};

class onlineSale : public sale{
  public:
  virtual void print(){
    cout << "Online Sale" << endl;
  }
};
void test(sale& arg1){
  //Polymorphism
  arg1.print();
}
int main(){
  sale s;
  discountSale ds;
  onlineSale os;

  test(s);
  test(ds);
  test(os);
  return 0;
}
```

Virtual Function: Summary

■ Virtual Function & Polymorphism

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

Overriding

- **Virtual function definition changed in a derived class**
  - We say it’s been "overidden"
  - Similar to *redefined* but only for virtual functions

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

MORE ABOUT VIRTUAL FUNCTIONS

C++11 `override` keyword

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

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

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

C++11 `override` keyword

- **`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

■override keyword

```cpp
#include <iostream>
using namespace std;

class A {
public:
    virtual void print() {
        cout << "A print" << endl;
    }
};

class B : public A {
public:
    void print(int a) {
        cout << "B print" << endl;
    }
};
```

```cpp
void test(A& arg1){
  //Polymorphism
  arg1.print();
}


int main()
{
    A a;
    B b;
    test(a);
    test(b);
    cout << "No Compiler Error" << endl;
    return 0;
}

```

C++11 `final` keyword

- **C++11 includes the `final` keyword**
  - to prevent a function from being overridden.
  - Useful if a function is overridden but don't want a derived classes to override it again.

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

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

final keyword

```cpp
#include <iostream>
using namespace std;

class A {
public:
    virtual void print() final {
        cout << "A print" << endl;
    }
    void test() {}
};

class B : public A {
public:
    void print() {
        cout << "B print" << endl;
    }
};
```

```cpp
int main()
{
    A a;
    B b;
    cout << "No Compiler Error" << endl;
    return 0;
}

```

Virtual Functions: Why Not All?

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

- **So <u>if virtual functions not needed, should not be used.</u>**

Pure Virtual Functions

- **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!
  - Make it a pure virtual function by adding **"=0"**:

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

Abstract Base Classes

- **Pure virtual functions require no definition**
  - Forces all derived classes to define "their own" version
- **Abstract base class (often, interface in other languages)**
  - Classes with one or more pure virtual functions
  - No objects can ever be created from it.
    - Since it doesn't have complete "definitions" of all it's members!
  - If derived class fails to define all pure's:
    - It's an abstract base class too

Pure virtual functions and abstract types

- Erroneous cases
  - Virtual functions without implementations in base classes
  - Creating objects of abstract types
  - Pure functions not having implementations even in derived classes

DEMO

```
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
{
  cout << "breed: " << breed << endl;
}
```

[Error Cases - Where are they?]

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

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

  dog.print();

  return 0;
}
```

EXTENDED TYPE COMPATIBILITY AND SLICING PROBLEM

Extended Type Compatibility

- **Given: Derived is derived class of Base**
  - 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 of 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!**
  - For example purposes only! Not typical!

Using Classes Pet and Dog

```
Dog vdog;
Pet vpet;

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

- **Anything that "is a" 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

- **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 not have "dog" properties
  - Makes for interesting philosophical debate.

■ Derived → Base, slicing problem example

```
#include <iostream>
using namespace std;

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

class Dog : public Pet
{
  public:
    Dog(){breed = "Great Dame";}
    string breed;
    virtual void print() const{cout << breed << endl;}
};

int main()
{
  Pet pet;
  Dog dog;
  pet = dog;
  //Pet pet2 = 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; it still "is a" Great Dane named Tiny

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

Resolving slicing problem using pointers

```
#include <iostream>
using namespace std;
class Pet{
  public:
    string name;
    virtual void print() const {cout << "Pet Class: " << name << endl;}
    Pet() {name = "pet"; }
};
class Dog : public Pet{
  public:
    Dog(){breed = "Great Dame";}
    string breed;
    virtual void print() const{cout << "Dog Class: " << breed << endl;}
};
int main()
{
  Pet* ppet;
  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!
  ppet->print();

  /*
  Pet p;
  Dog d;
  d.name = "Jason";
  d.breed = "Jack";
  p = d;
  p.print();
  */
  return 0;

```

Virtual Destructors

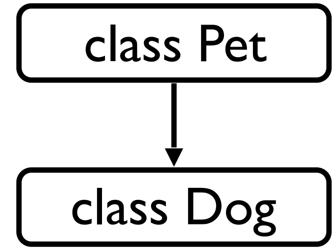
- **Recall:**
  - destructors needed to de-allocate dynamically allocated data
- **Consider:**

```
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**

Upcasting and Downcasting



Consider:

```
Pet vpet;
Dog vdog;
…
vdog = static_cast<Dog>(vpet);  // ILLEGAL!
```

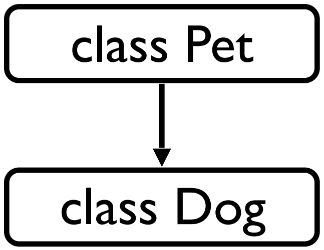
- No downcasting: From ancestor type to descendant type

Can’t cast a pet to be a dog, but:

```
vpet = vdog;   // Legal!
vpet = static_cast<Pet>(vdog);  // Also legal!
```

- Upcasting is OK: From descendant type to ancestor type

Downcasting with `dynamic_cast`



- **Downcasting dangerous!**
  - Casting from ancestor type to descended type.
  - Assumes information is "added“.
  - Can be done with dynamic_cast:

```
Pet *ppet;
ppet = new Dog;
Dog *pdog = dynamic_cast<Dog*>(ppet);
```

    - Legal, but dangerous!

- **Downcasting rarely done due to pitfalls.**
  - Must track all information to be added.
  - All member functions must be virtual.

Summary

- Virtual Function
- Polymorphism
- Override, Final
- Pure Virtual Function
- Abstract Class
- Slicing Problem
- Virtual Destructor
- Upcasting & Downcasting