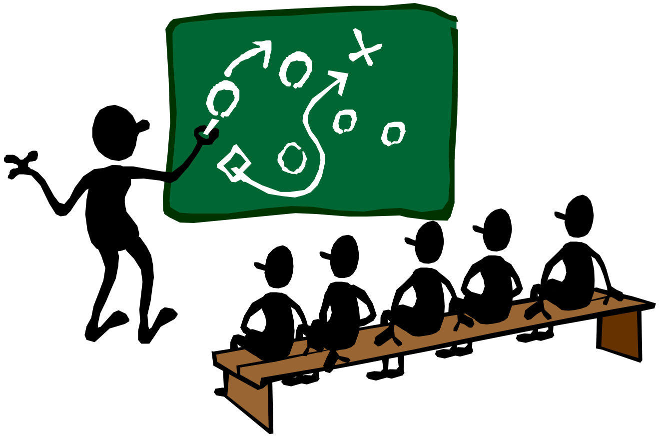
C++ Programming Language Chapter 15 Polymorphism and Virtual Functions



Juinn-Dar Huang
Associate Professor
jdhuang@mail.nctu.edu.tw

May 2011

Learning Objectives

- Virtual functions
- Polymorphism
- Abstract classes and pure virtual functions
- How to design a good class hierarchy

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Member Function Redefinition (1/2)

- In the previous chapter,

```
class Employee {
    // data members
public:
    void print() const;  // Employee::print()
    // other member functions
};

class Manager : public Employee {
    // data members
public:
    void print() const;  // Manager::print(),
                         // Manager REDIFINES Employee's print()
    // other member functions
};
```

Juinn-Dar Huang   jdhuang@mail.nctu.edu.tw

Member Function Redefinition (2/2)

```
void Employee::print() const {
    cout << family_name << '\t' << department << endl;
     /* … */ }
void Manager::print() const {
    Employee::print();                       // explicitly call Employee's print();
    cout << "Level: "  << level << endl;  // that's ok, since Manager is an Employee
    /* … */ }
void f() {
    Employee cl("Chi-Ling", 3);  Manager adar("Adar", 3, 1);
    cl.print();                  // use Employee::print()
    adar.print();                // use Manager::print()
    Employee* pe = &cl; Manager* pm = &adar;
    pe->print();                 // use Employee::print()
    pm->print();                 // use Manager::print()
    pe = &adar;                  // ok! adar is also an Employee
    pe->print();                 // use Employee::print(); but adar is a manager …☹
}
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Polymorphism

Virtual Functions (1/2)

- Is there any chance to fix it? ➔ Yes, there is!
- Add one keyword at …

```
class Employee {
    // data members;
public:
    virtual void print() const;          // Employee::print() is now a virtual function
    // other member functions            // once a function is declared virtual,
};                                       // it's ALWAYS virtual in all derived classes

class Manager : public Employee {
    // data members;
public:
    void print() const;                  // Manager::print()
    // other member functions            // Manager's print OVERRIDES Employee's print
};
```

"virtual" is optional here (preferred not)

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Polymorphism

Virtual Functions (2/2)

```
void Employee::print() const {
    cout << family_name << '\t' << department << endl;
    /* … */ }
void Manager::print() const {
    Employee::print();        // explicitly call Employee's print()
    cout << "Level: "  << level << endl;
    /* … */ }
```

unchanged at all

No "virtual" there! Otherwise, error!

```
void f() {
    Employee cl("Chi-Ling", 3);  Manager adar("Adar", 3, 1);
    Employee* pe = &cl; Manager* pm = &adar;
    pe->print();              // use Employee::print()
    pm->print();              // use Manager::print()
    pe = &adar;               // ok! adar is also an Employee
    pe->print();              // use Manager::print()! Program knows what
}                             // pe points to is actually a Manager ☺, What a magic!
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Virtual vs. Non-Virtual Functions (1/2)

- For non-virtual (member) functions
  - function calls are **STATICALLY** bound (i.e., bound at compile time)

```
class B {                               class D : public B {
public:                                 public:
    void mf();                          void mf();      // redefine mf();
};                                      };

void f() {
    B b, *pB = &b;   D d, *pD = &d;
    b.mf();        // statically binding ➔ b is of type B ➔ call B::mf()
    d.mf();        // statically binding ➔ d is of type D ➔ call D::mf()
    pB->mf();      // statically binding ➔ pB is of type B* ➔ call B::mf()
    pD->mf();      // statically binding ➔ pD is of type D* ➔ call D::mf()
    pB = &d;       // ok, D is derived from B
    pB->mf();      // still statically binding ➔ pB is of type B* ➔ call B::mf()
}                  // though pB actually points to d (an object of type D)
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Virtual vs. Non-Virtual Functions (2/2)

- For virtual (member) functions
  - must be non-static member functions
  - function calls are **DYNAMICALLY** bound (i.e., bound at runtime) if they are invoked through **pointers** or **references**

```
class B {                          class D : public B {
public:                            public:
    virtual void mf(); };          void mf();        // override B::mf(); };

void f() {
    B b, *pB = &b;   D d, *pD = &d;
    b.mf();       // still statically binding ➔ b is of type B ➔ call B::mf()
    d.mf();       // still statically binding ➔ d is of type D ➔ call D::mf()
★   pB->mf();     // dynamically binding ➔ pB actually points to b ➔ call B::mf()
    pD->mf();     // dynamically binding ➔ pD actually points to d ➔ call D::mf()
    pB = &d;      // ok, D is derived from B
★   pB->mf();     // dynamically binding ➔ pB actually points to d ➔ call D::mf()
}
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Power of Virtual Functions

```
void g(B* pB, B& rB) {
    pB->mf();        // call B::mf() or D::mf()?
    rB.mf();         // call B::mf() or D::mf()?
    // unknown at compile time;  dynamically bound; determined at runtime
}
```

- Few years later, you decide to add a new derived class E

```
class E : public D {   // E is publicly inherited from D
public:
    void mf();           // override D::mf()
};
void h() {  E e; g(&e, e);  }
```

  - guess what? ➔ in g(&e, e) call, E::mf() is invoked!
  - the best part is ➔ no need to recompile class B, class D and g()

Juinn-Dar Huang   jdhuang@mail.nctu.edu.tw

Polymorphism

- Polymorphism
  - while accessing a member function, the correct version based on the actual calling object is always invoked
  - namely, the behavior of calling a member function through a pointer/reference may be different ➔ polymorphic
- In C++, polymorphism is achieved through
  - virtual functions, and
  - manipulating objects through pointers or references
- A class with virtual functions is called a polymorphic class
- Polymorphism is another cornerstone of OOP

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Redefine vs. Override

- When a derived class D **modifies** the definition of an inherited non-virtual member function mf
  - we say class D redefines mf, or mf is redefined in D

  class B { public: void mf(); }

  class D: public B { public: void mf(); } // D redefines mf()

- When a derived class D **modifies** the definition of a virtual member function mf inherited from class B
  - we say D::mf overrides B::mf, or B::mf is overridden by D::mf

  class B { public: virtual void mf(); }

  class D: public B { public: void mf(); } // D::mf() overrides B::mf()

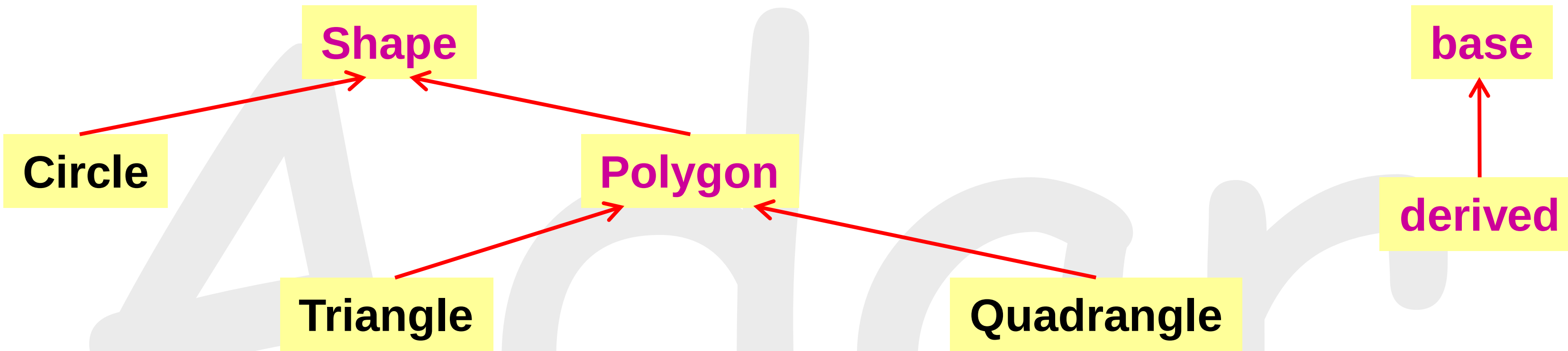
- Fundamental conceptual differences between them
  - details will be given later

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Concrete Class vs. Abstract Class (1/2)

- We've learned that a class represents a concept
- Some concepts are concrete and some are abstract



- Abstract classes: Shape and Polygon
  - e.g., no idea how to draw or rotate an arbitrary shape
  - objects of abstract classes should not exist (they are abstract)
- Concrete classes : Circle, Triangle and Quadrangle
  - objects of these types can exist
  - they can be drawn, rotated, …

Concrete Class vs. Abstract Class (2/2)

- One way to implement an abstract class

```
class Shape {
public:
    virtual void rotate(int) { cerr << “Cannot rotate a shape\n”; }
    virtual void draw() { cerr << “Cannot draw a shape\n”; }
    // …
};

void f() {
    shape s;        // legal but silly; a shapeless shape object
    s.rotate(90);   // error message; cannot rotate a shapeless shape object
    s.draw();       // error message; cannot draw a shapeless shape object
}
```

- Any better implementation?

Pure Virtual Functions & Abstract Class

- How to correctly implement an abstract class in C++?

```
class Shape {
public:
    virtual void rotate(int) = 0;        // pure virtual function
    virtual void draw() = 0;             // pure virtual function
    virtual bool is_closed() = 0;        // pure virtual function
    // …                                 // only declaration; no definition
};
void f() {
    Shape s;        // compilation error! it must be an error, or
    // s.draw();    would be legal ; but draw() is a pure virtual function
};
```

- A class with one or more pure virtual functions is called an abstract class
- No objects of abstract class can be created in C++

Abstract Base Class (ABC) (1/3)

- An abstract class can be used only
  - as a base class for other classes ➔ abstract base class (ABC)
  - namely, as an interface specification

```
class Point { /* define a point in a 2D space */ };
class Circle : public Shape {
public:
    void rotate(int) {  }                // override pure Shape::rotate
    void draw();                         // override pure Shape::draw
    bool is_closed() { return true; }    // override pure Shape::is_closed
    Circle(Point center, double r);      // ctor
private:
    Point center;  double radius;
};
void f() {
    Circle c(Point(4.0, 5.0), 3.0);      // Circle is a concrete class now
    c.draw();                            // Yes, a Circle object can be drawn!
}
```

Abstract Base Class (ABC) (2/3)

- A derived class becomes concrete once it overrides **ALL** inherited pure virtual functions
  - e.g., just like Circle

Two key notions here

- Abstract class is always used as a base class ➔ (ABC)
  - you cannot create objects of abstract class
  - it only makes sense that some classes derived from it and become concrete by overriding all pure functions
- Abstract class specifies interface requirements
  - a class D derived from an ABC B must override all pure virtual functions of B to become concrete
  - it implies that D has no choice but provides definitions for all those pure virtual functions specified by B to become concrete

Abstract Base Class (ABC) (3/3)

- A class derived from an ABC is still abstract if it doesn't override **ALL** inherited pure virtual functions
  - the following Polygon is still an abstract class

```
class Polygon : public Shape {
public:
    bool is_closed() { return true; }    // override Shape::is_closed
    // draw & rotate not overridden ➔ Polygon is still abstract
};

class Triangle : public Polygon {
public:
    void draw();              // override Shape::draw
    void rotate(int);         // override Shape::rotate
    // …                      // Now, Triangle becomes a concrete class!
};                            // i.e., objects of Triangle can be created!
```

Why Abstract Base Class?

- What kind of nut wants to define a class that cannot be used to create objects?

```
void draw_shapes(Shape* sarr[], int size) {
    for(int i = 0; i < size; ++i)
        sarr[i]->draw();   // objects of Circle, Triangle, Quadrangle, …
}
```

- draw_shapes can **correctly** draw **ALL** kinds of objects of concrete classes derived from Shape
  - like discussion on Page 8, new concrete classes can be added and draw_shapes can still work correctly w/o the need of recompilation
- Without Shape, it is impossible to manipulate objects of Circle and Triangle through a same type of pointer (Shape*)
- That's exactly why we need ABC!

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Public Inheritance

- In C++, public inheritance implies "is-a" relationship
  - derived class inherits all data members from base classes
  - derived class inherits all non-private member functions from base classes
  - every derived class object IS a bass class object (polymorphically through pointer/reference)

- So, **make sure public inheritance models "is-a"** when you are using it

// Do **NOT** do things like…

class Employee : public Manager { /* … */ }; // Every employee is a manger?!

class Quadrangle : public Triangle { /* … */ }; // What?! Who taught you math?

Avoid Hiding Inherited Names (1/3)

- Name hiding issue (we've discussed this in Chapter 3)

```
int x;      // global x
void f() {
    double x;       // x hides x even if they are of different types!
    cin >> x;
}
```

- Same scope resolution rule applies to inheritance
  ➔ names in derived class hide those in base classes

Avoid Hiding Inherited Names (2/3)

```
class Base { public:
    virtual void mf1() = 0;     // pure virtual function
    virtual void mf1(int);      // overloaded simple virtual function
    virtual void mf2();         // simple virtual function
    void mf3();                 // non-virtual member function
    void mf3(double);           /* overloaded non-virtual member function */   };

class Derived : public Base { public:
    virtual void mf1();         // override Base::mf1
    void mf3();                 /* redefine mf3 */   };

void f() {
    Derived d;
    d.mf1();                    // ok, call Derived::mf1()
    d.mf1(10);                  // surprising error! Derived::mf1 hides Base::mf1
    d.mf2();                    // ok, call Base::mf2()
    d.mf3();                    // ok, call Derived::mf3()
    d.mf3(10.0);                // surprising error! Derived::mf3 hides Base::mf3
}
```

Avoid Hiding Inherited Names (3/3)

```
class Base {
    // same stuffs here …
};

class Derived : public Base {
public:
    using Base::mf1;        // make all things in Base named mf1 visible in Derived
    virtual void mf1();     // override Base::mf1
    void mf3();             /* redefine mf3 */
};

void f() {
    Derived d;
    d.mf1();                // ok, call Derived::mf1()
    d.mf1(10);              // ok now! call Base::mf1(int)
    d.mf2();                // ok, call Base::mf2()
    d.mf3();                // ok, call Derived::mf3()
    d.Base::mf3(10.0);      // ok now! call Base::mf3(double) explicitly
}
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Understand What You Are Saying …

There are several ways for classifying member functions

- Member functions can be private, protected and private
  - different level of access control
- Member functions can be static and non-static
  - static: without implicit this pointer
  - non-static: with implicit this pointer
- A non-static member function can be constant or not
  - constant one guarantees not to modify the calling object

- A non-static member functions can be a
  - pure virtual function
  - simple virtual function
  - non-virtual function

Polymorphism-related

Interface vs. Implementation Inheritance

At first, **member function interfaces** are always inherited

- While declaring a member function pure virtual
  - intent: to have derived classes inherit a function interface **ONLY**
  - derived classes **MUST** re-declare it simple virtual and provide actual definition to become concrete classes
- While declaring a member function simple virtual
  - intent: to have derived classes inherit a function interface as well as a **DEFAULT** implementation
  - derived classes can choose to use same default implementation, or
  - they can **OVERRIDE** the default implementation for specialization
- While declaring a member function non-virtual
  - intent: to have derived classes inherit a function interface as well as a **MANDATORY** implementation
  - non-virtual function specifies an **INVARIANT** over specialization, and thus should **NEVER** be **REDEFINED**

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Common Pitfalls

Common pitfalls for beginners

- Always declare all member functions non-virtual in a class
  - it is correct only if this class intends not to be inherited at all
  - or, there are no rooms for specialization in derived classes
- Always declare all member functions virtual in all classes
  - yes, abstract base classes may declare all member functions virtual
  - however, concrete base classes usually have certain invariants

Inherited Non-Virtual Functions

```
class B {
public:
    // mf is a non-virtual function ➔ specifies an invariant over specialization
    void mf() const {  cout << "This is an invariant defined by B\n";  }
};

class D : public B {
public:
    // REDIFINE mf here ➔ violate the advice given in the previous slide
    void mf() const {  cout << "Who cares?\n";  }

void f() {
    D d, *pD = &d;          // pD points to d
    B *pB = &d;             // pB also points to d
    pD-> mf();              // call D::mf();
    pB->mf();               // call B::mf();  // punishment for not obeying the advice
}
```

- **Never redefine inherited non-virtual functions**

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Polymorphism

Virtual Destructors (1/2)

```
class Base {
public:
    ~Base();                          // non-virtual dtor
    // other stuffs
};

class Derived: public Base { /* add some data members … */ };

void f() {
    Base* pB = new Derived;           // ok, get right size of memory,
    // …                              // then call Derived’s ctor
    delete pB;                        // Disaster! call Base’s dtor since it’s non-virtual
                                      // ➔ wrong size of memory gets returned
}
```

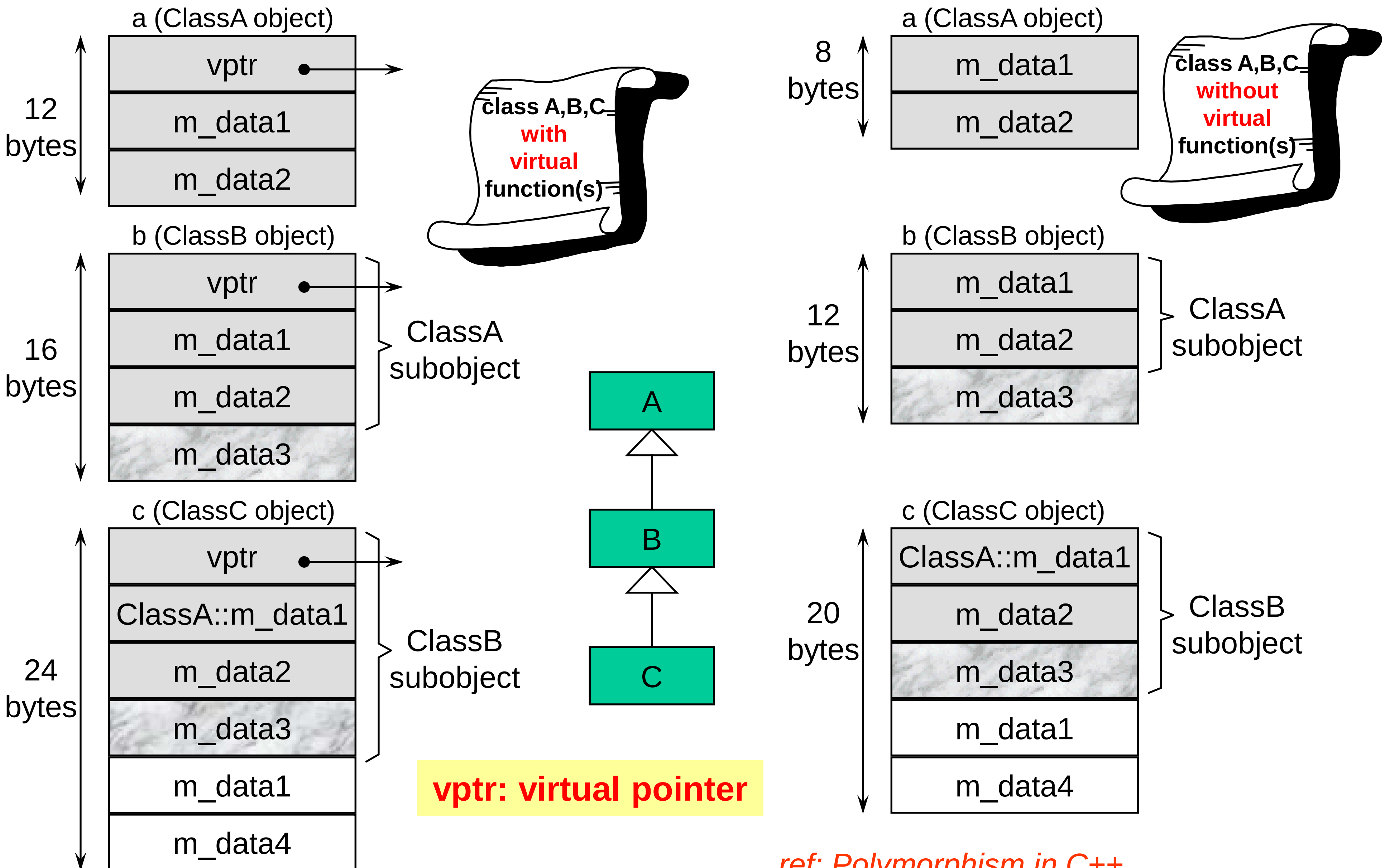
Virtual Destructors (2/2)

```
class Base {
public:
    virtual ~Base();                    // virtual dtor now
    /* other stuffs */    };
class Derived: public Base { /* add some data members … */ };
void f() {
    Base* pB = new Derived;             // ok, get right size of memory,
    // …                                // then call Derived's ctor
    delete pB;                          // ok! call Derived's dtor since it's virtual
                                        // ➔ right size of memory gets returned
}
```

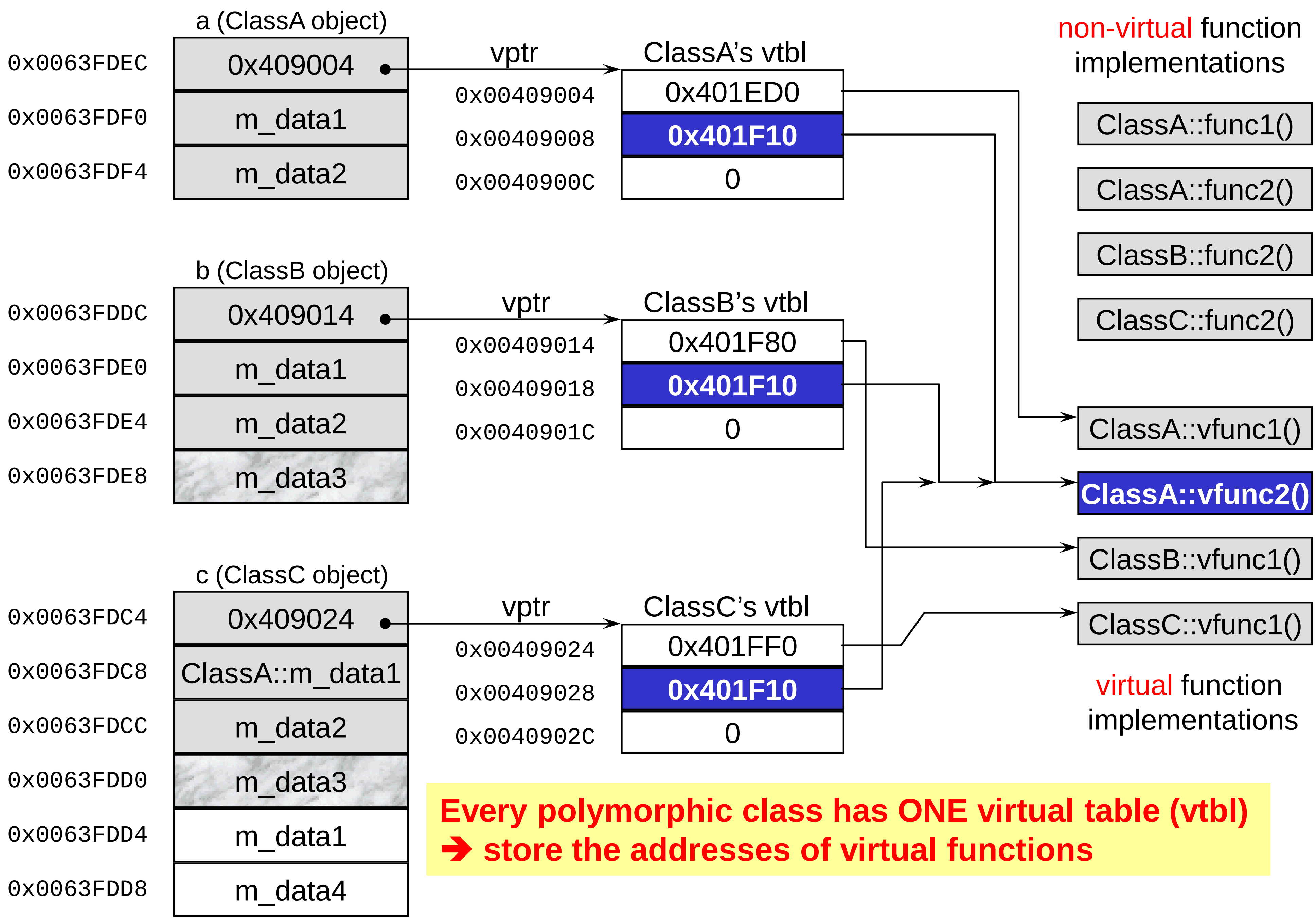
- **Declare dtors virtual in polymorphic base classes**
  - polymorphic base ➔ has at least one virtual function (excluding dtor)
- Don't blindly declare dtors virtual in all classes
  - incurs memory and runtime overhead ➔ there is no free lunch
  - ugly truth: polymorphism comes with costs (see next 3 slides)

How does polymorphism work?

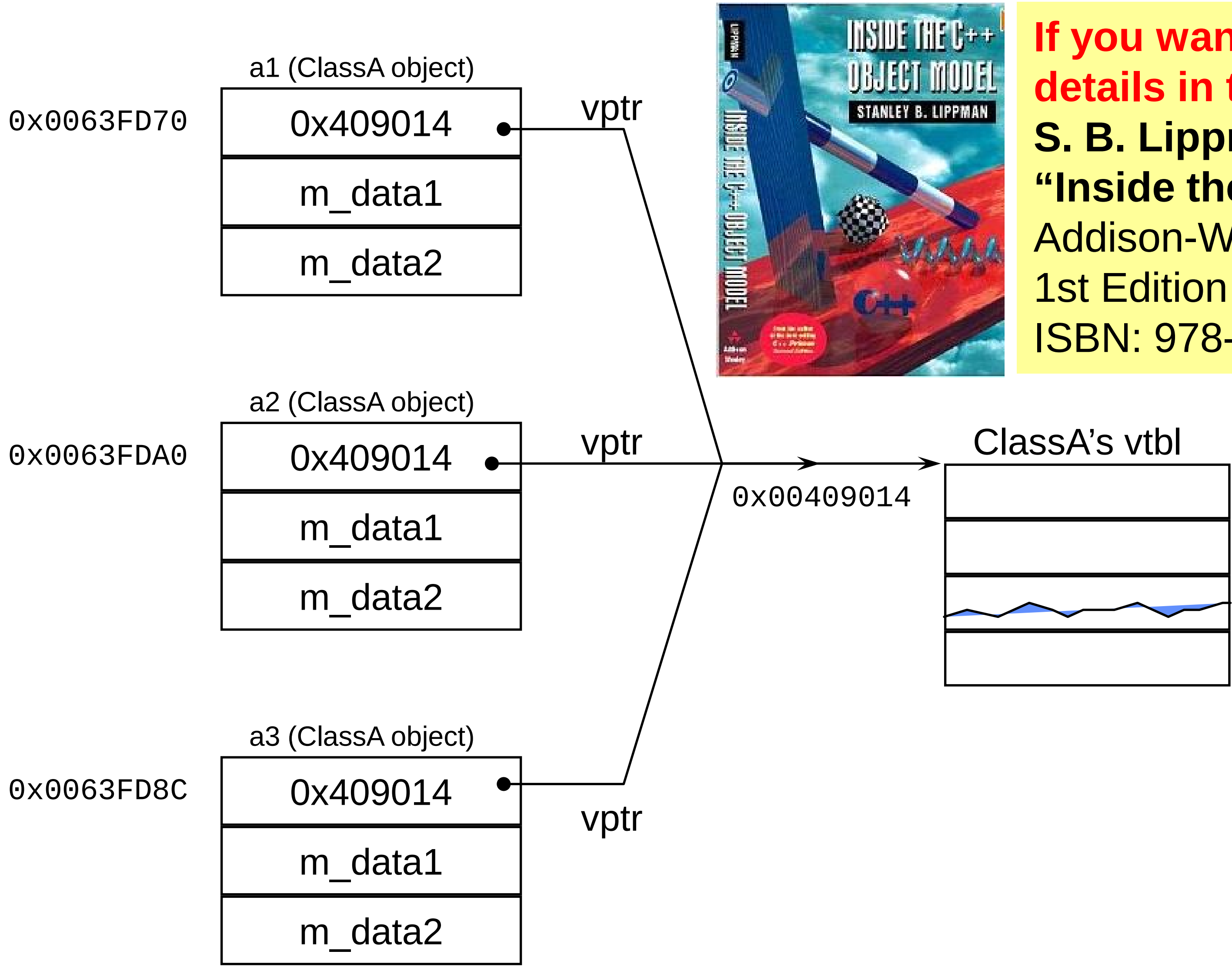
Advanced



ref: Polymorphism in C++



Every polymorphic class has ONE virtual table (vtbl)
➔ store the addresses of virtual functions



If you want to learn more details in this topic ➔
S. B. Lippman,
"Inside the C++ Object Model,"
Addison-Wesley Professional;
1st Edition (May 13, 1996),
ISBN: 978-0201834543

Again, every polymorphic class has exactly ONE virtual table (vtbl)
➔ store the addresses of virtual functions

Summary

- Virtual functions vs. non-virtual functions
  - dynamically binding vs. statically binding
- Polymorphism by virtual functions
- Concrete classes vs. abstract classes
- Abstract base classes by pure virtual functions
- Inheritance of interface vs. inheritance of implementation
  - pure virtual functions
  - virtual functions
  - non-virtual functions
- A bunch of advices save you from lots of troubles
  - shown in the next slide

Advices

- Make sure public inheritance models “is-a”
- Model “has-a” through composition
- Avoid hiding inherited names
- Differentiate between
  inheritance of interface and inheritance of implementation
  - pure virtual function vs. simple virtual function vs. non-virtual function
  - say what you mean; understand what you are saying!
- Never redefine inherited non-virtual functions
- Declare destructors virtual in polymorphic base classes