

Chapter 15

Polymorphism and Virtual Functions

Polymorphism



Learning Objectives

- 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

- 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

- Best explained by example:
- Classes for several kinds of figures
 - Rectangles, circles, ovals, etc.
 - Each figure an object of different class
 - Rectangle data: height, width, center point
 - Circle data: center point, radius
- All derive from one parent-class: Figure
- Require function: draw()
 - Different instructions for each figure

Figures Example 2

- Each class needs different *draw* function
- Can be called "draw" in each class, so:
Rectangle r;
Circle c;
r.draw(); //Calls Rectangle class's draw
c.draw(); //Calls Circle class's draw
- Nothing new here yet...

Figures Example: center()

- Parent class Figure contains functions that apply to "all" figures; consider:
center(): moves a figure to center of screen
 - Erases 1st, then re-draws
 - So Figure::center() would use function draw() to re-draw
 - Complications!
 - Which draw() function?
 - From which class?

Figures Example: New Figure

- Consider 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"!

Figures Example: Virtual!

- Virtual functions are the answer
- Tells compiler:
 - "Don't know how function is implemented"
 - "Wait until used in program"
 - "Then get implementation from object instance"
- Called **late binding** or **dynamic binding**
 - Virtual functions implement late binding

Virtual Functions: Another Example

- Bigger example best to demonstrate
- Record-keeping program for automotive parts store
 - Track sales
 - Don't know all sales yet
 - 1st only regular retail sales
 - Later: Discount sales, mail-order, etc.
 - Depend on other factors besides just price, tax

Virtual Functions: Auto Parts

- 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

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

Member Functions

savings and operator <

- `double Sale::savings(const Sale& other) const`
`{`
 `return (bill() – other.bill());`
`}`
- `bool operator < (const Sale& first,`
`const Sale& second)`
`{`
 `return (first.bill() < second.bill());`
`}`
- Notice BOTH use member function `bill()`!

Class Sale

- Represents sales of single item with no added discounts or charges.
- Notice reserved word "virtual" in declaration of member function *bill*
 - Impact: Later, derived classes of Sale can define THEIR versions of function bill
 - Other member functions of Sale will use version based on object of derived class!
 - They won't automatically use Sale's version!

Derived Class DiscountSale Defined

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

# DiscountSale's Implementation of bill()

- ```
double DiscountSale::bill() const  
{  
    double fraction = discount/100;  
    return (1 – fraction)*getPrice();  
}
```
- Qualifier "virtual" does not go in actual function definition
 - "Automatically" virtual in derived class
 - Declaration (in interface) not required to have "virtual" keyword either (but usually does)

DiscountSale's Implementation of bill()

- Virtual function in base class:
 - "Automatically" virtual in derived class
- Derived class declaration (in interface)
 - Not required to have "virtual" keyword
 - But typically included anyway, for readability

Derived Class DiscountSale

- DiscountSale's member function bill() implemented differently than Sale's
 - Particular to "discounts"
- Member functions *savings* and "<"
 - Will use this definition of bill() for all objects of DiscountSale class!
 - Instead of "defaulting" to version defined in Sales class!

Virtual: Wow!

- Recall class Sale written long before derived class DiscountSale
 - Members savings and "<" compiled before even had ideas of a DiscountSale class
- Yet in a call like:
DiscountSale d1, d2;
d1.savings(d2);
 - Call in savings() to function bill() knows to use definition of bill() from DiscountSale class
- Powerful!

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

- Virtual function definition changed in a derived class
 - We say it's been "overridden"
- Similar to redefined
 - Recall: for standard functions
- So:
 - Virtual functions changed: ***overridden***
 - Non-virtual functions changed: ***redefined***

Virtual Functions: Why Not All?

- Clear advantages to virtual functions as we've seen
- One major disadvantage: overhead!
 - Uses more storage
 - Late binding is "on the fly", so programs run slower
- So if virtual functions not needed, should not be used

Pure Virtual Functions

- Base class might not have "meaningful" definition for some of its members!
 - Its 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:
`virtual void draw() = 0;`

Abstract Base Classes

- Pure virtual functions require no definition
 - Forces all derived classes to define "their own" version
- Class with one or more pure virtual functions is: abstract base class
 - Can only be used as base class
 - 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

Extended Type Compatibility

- Given:
Derived is derived class of Base
 - Derived objects can be assigned to objects of type Base
 - But NOT the other way!
- 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;
};
```

# Classes Pet and Dog

- Now given declarations:  
Dog vdog;  
Pet vpet;
- Notice member variables name and breed are public!
  - For example purposes only! Not typical!

# Using Classes Pet and Dog

- Anything that "is a" dog "is a" pet:
  - `vdog.name = "Tiny";`  
`vdog.breed = "Great Dane";`  
`vpet = vdog;`
  - 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!
  - `cout << vpet.breed;`
    - Produces ERROR msg!
  - Called slicing problem
- 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

# Slicing Problem Fix

- In C++, slicing problem is nuisance
  - It still "is a" Great Dane named Tiny
  - We'd like to refer to it's breed even if it's been treated as a Pet
- Can do so with pointers to dynamic variables

# 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!
```

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

Slicing Problem Example (Cont.)

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

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

Virtual Destructors

- Recall: destructors needed to de-allocate dynamically allocated data
- Consider:
Base *pBase = new Derived;
...
delete pBase;
 - 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

Casting

- Consider:
Pet vpet;
Dog vdog;
...
vdog = static_cast<Dog>(vpет); //ILLEGAL!
- Can't cast a pet to be a dog, but:
vpет = vdog; // Legal!
vpет = static_cast<Pet>(vdog); //Also legal!
- Upcasting is OK
 - From descendant type to ancestor type

Downcasting

- 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

Inner Workings of Virtual Functions

- Don't need to know how to use it!
 - Principle of information hiding
- Virtual function table
 - Compiler creates it
 - Has pointers for each virtual member function
 - Points to location of correct code for that function
- Objects of such classes also have pointer
 - Points to virtual function table

Summary 1

- Late binding delays decision of which member function is called until runtime
 - In C++, virtual functions use late binding
- Pure virtual functions have no definition
 - Classes with at least one are abstract
 - No objects can be created from abstract class
 - Used strictly as base for others to derive

Summary 2

- Derived class objects can be assigned to base class objects
 - Base class members are lost; slicing problem
- Pointer assignments and dynamic objects
 - Allow "fix" to slicing problem
- Make all destructors virtual
 - Good programming practice
 - Ensures memory correctly de-allocated