CSE 230 Intermediate Programming in C and C++

Polymorphism and Virtual Functions

Fall 2017

Stony Brook University

Instructor: **Shebuti Rayana**

http://www3.cs.stonybrook.edu/~cse230/

Ref. Book: C How to Program, 8th edition by Deitel and Deitel

Polymorphism

- It is possible to design and implement systems that are more easily *extensible* through polymorphism.
- Polymorphism enables you to write more general programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy's base class.
- Classes that do not exist during program development can be added with little or no modification to the generic part of the program
  – as long as those classes are part of the hierarchy that is being processed generically.
- The only part of the program that will need modification are those parts that require direct knowledge of the particular class that is added to the hierarchy.

Polymorphism: Example

- One means of dealing with objects of different types is to use polymorphism.
- For example, in a hierarchy of shapes in which each shape specifies its type as a data member, with polymorphism the compiler could determine which **print** function to call based on the type of the particular object.
- Polymorphism and **virtual** functions can eliminate the need for **switch** logic. It also avoids errors typically associated with equivalent **switch** logic and facilitates testing, debugging and program maintenance.
- Suppose a set of shape classes such as **Circle**, **Triangle**, **Square**, etc. are all derived from base class **Shape** and each class includes a separate **draw** function.

Virtual Functions

- To draw any shape, we could simply call function **draw** of base class **Shape** and let the program determine which derived class **draw** function to use.
- To enable this kind behavior, **draw** must be declared in the base class (i.e. **Shape**) as a **virtual** function and then each derived class overrides **draw** to draw the appropriate shape.
- A **virtual function** is a member **function** that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a **virtual function** for that object and execute the derived class's version of the **function**.
- Following may appear in **Shape**:

virtual void draw() const;

Virtual Functions (cont.)

- Once a function is declared **virtual**, it remains **virtual** all the way down the inheritance hierarchy from that point even if it is not declared **virtual** when a class overrides it. However, explicit declaration promotes clarity.

Abstract and Concrete Classes

- A class is made **abstract** by declaring one or more of its **member** functions to be virtual without definition.
- No objects of an abstract base class can be instantiated.
- A **pure virtual** function is one with an initializer of **= 0** in its declaration as in:

  virtual void draw() const = 0;
- If no definition is supplied in a derived class for a pure **virtual** function, then the function remains to be pure. Consequently, the derived class is also an abstract class.

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"

Shapes Example

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

Shapes 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…

Shape Example: center()

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

Shape Example: New Shape

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

Shapes 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
  - Binding is done at run time

Virtual Functions: Why Not All?

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

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

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