(7-2) Classes: A Deeper Look
D & D Chapter 9

Instructor - Andrew S. O'Fallon
CptS 122 (February 23, 2024)
Washington State University

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Key Concepts

- Composition relationship
- `const` objects
- `const` member functions
- The "`this`" pointer

Composition Relationship

- A class can have objects of other classes as members – this is composition
- Composition is also referred to as a *has-a* relationship (we will not distinguish between composition and aggregation at this point)
  - For example: a car *has-an* engine, a pencil *has-an* eraser, etc.

`const` Objects

- Some objects need to be *mutable* and some do not (*immutable*)
  - A *mutable* object's attributes may be modified (given different values) after creation of the object
  - An *immutable* object's attributes have to be set during construction and cannot be modified later
    - Objects can be declared as immutable using keyword `const`
    - For example, consider a *ComplexNumber* with an imaginary and real part:

```
ComplexNumber c1(2.5, 3.0) // mutable
const ComplexNumber c2(4.5, 6.0); // immutable
```

`const` Member Functions

- Getter/accessor functions in most cases should be declared as `const` member functions
  - For example:

    double getRealPart () const; // declaration in ComplexNumber
- `const` member function cannot modify members of the object
  - They also *cannot* call functions that try to modify members of the object
- NOTE: `const` objects *cannot* call non-`const` member functions!!! However non-`const` objects can call `const` member functions

Copy Constructors for `const` Objects

- How do we copy a `const` object?
  - We could use a copy constructor where the argument is a reference to a `const` object
  - ComplexNumber (`const` ComplexNumber &copy);
- For example:

```
const ComplexNumber c2(4.5, 6.0); // immutable
ComplexNumber c3(c2); // invokes the copy constructor with the const argument
ComplexNumber c4 = c3; // will actually invoke the copy constructor, not overloaded
                       // assignment because we are constructing (instantiating)
                       // an object here!
```


The "`this`" Pointer (1)

- Every object has access to a *pointer* called keyword `this`
- It stores the address of the object
- The pointer is not part of the object itself, but is an *implicit* argument (passed by the compiler) to each of the object's *non-static* member functions
- It can be used *explicitly* to reference data members in order to avoid name *conflicts*

The "`this`" Pointer (2)

- Let's say we named one of the private data members of class ComplexNumber *realPart*:

  private:

  ```
  double realPart; // of course we'll generally name mRealPart
  ```

- We want to create a setter for the *realPart*. We need to avoid *ambiguous* statements!:

  public:

  ```
  void setRealPart (double realPart)
  {
          realPart = realPart; // ambiguous statement!
          this->realPart = realPart; // use " this" explicitly instead!
  }
  ```


Type of "`this`" Pointer

- The type is dependent on the type of object
- For a non-`const` member function of *ComplexNumber*, the `this` pointer type would be *ComplexNumber* *
  - For a `const` member function, the `this` pointer type would be `const` ComplexNumber * -- meaning it could not be used to modify members of the object!

References

- P.J. Deitel & H.M. Deitel, *C++: How to Program* (9th ed.), Prentice Hall, 2014
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

Collaborators

- Jack Hagemeister