(7 - 2) Operator Overloading D & D Chapter 10

Instructor - Andrew S. O'Fallon

CptS 122 (March 3, 2021)

Washington State University

Key Concepts

- Keyword `operator`
- Operator overloading

A. O'Fallon, J. Hagemeister

What is Operator Overloading?

- A generalization of function overloading

- An extension to C++ standard operators to define how they should work with user-defined types such as classes

A. O'Fallon, J. Hagemeister

Why Overload Operators?

- Improves readability
- Allows for a more natural way to implement code

A. O'Fallon, J. Hagemeister

Rules and Restrictions on Operator Overloading

- The precedence of an operator cannot be changed

- The associativity of an operator cannot be changed, i.e. left-to-right or right-to-left

- The "arity" of an operator cannot be changed, i.e. if the operator accepts one operand (unary) or two operands (binary)

- Only existing operators may be overloaded

A. O'Fallon, J. Hagemeister

Which Operators Cannot be Overloaded?

- .
- .* (pointer to member)
- ::
- ?:

A. O'Fallon, J. Hagemeister

Recall Class ComplexNumber's Add () Function

- Let's write the definition for the `add()` member function

```
// Prototype: ComplexNumber add (const ComplexNumber &operand);

// Definition – notice the binary scope resolution operator
ComplexNumber ComplexNumber::add (const ComplexNumber &operand)
{
        // This adds the real part of the "operand" object
        // to the real part of the object that invokes the
        // call to the function; it also adds the imaginary
        // parts
        ComplexNumber result; // Declare local ComplexNumber

        // Recall we use the dot member operator (.) to access
        // members of a class; no dot (.) denotes accessing the
        // instantiated object's members; note we don't have to apply "special"
        // operators to access an object passed by reference!
        result.mRealPart = mRealPart + operand.mRealPart;
        result.mImaginaryPart = mImaginaryPart + operand.mImaginaryPart;
         // Don't want to pass back by reference; cause undefined behavior
        return result;
}
```

We Can Replace Add () by Overloading + (Using a Friend Function)

- Let's write a function to overload the binary + operator; this function is a *non-member* function, but is a friend of `ComplexNumber`

```
// Prototype: friend ComplexNumber operator+ (const ComplexNumber &lhs,
                                              const ComplexNumber &rhs);


// Definition – notice the operator is not preceded by ComplexNumber::, because
// it's a non-member operator. Made this function a friend of ComplexNumber
// to efficiently access the private data members of ComplexNumber.
ComplexNumber operator+ (const ComplexNumber &lhs, const ComplexNumber &rhs)
{
        ComplexNumber result; // Declare local ComplexNumber

        result.mRealPart = lhs.mRealPart + rhs.mRealPart;
        result.mImaginaryPart = lhs.mImaginaryPart + rhs.mImaginaryPart;

         // Don't want to pass back by reference; cause undefined behavior
        return result;
}
```

A. O'Fallon, J. Hagemeister

We Can Replace Add () by Overloading + (without Using a Friend Function)

- Let's write a function to overload the binary + operator; this function is a *non-member* function, but is NOT a friend of `ComplexNumber`

```
// Prototype: ComplexNumber operator+ (const ComplexNumber &lhs,
                                       const ComplexNumber &rhs);


// Definition – notice the operator is not preceded by ComplexNumber::, because
// it's a non-member operator. This function is NOT a friend of ComplexNumber.
// Need to use setters/getters now!
ComplexNumber operator+ (const ComplexNumber &lhs, const ComplexNumber &rhs)
{
        ComplexNumber result; // Declare local ComplexNumber

        result.setRealPart (lhs.getRealPart() + rhs.getRealPart());
        result.setImaginaryPart (lhs.getImaginaryPart() +
                                 rhs.getImaginaryPart());
         // Don't want to pass back by reference; cause undefined behavior
        return result;
}
```

A. O'Fallon, J. Hagemeister

Why Non-Member Overloaded Operators?

- Enables "symmetry" and *communitivity* among operators, i.e.

  - ```
    ComplexNumber operator+ (const
    ComplexNumber &lhs, int rhs);
    ```

  - ```
    ComplexNumber operator+ (int lhs,
    const ComplexNumber &rhs);
    ```

    - ```
      Important to make non-member operator+
      when lhs is not a class! Since lhs is not
      an object in this case!
      ```

A. O'Fallon, J. Hagemeister

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

A. O'Fallon, J. Hagemeister

Collaborators

- Jack Hagemeister

A. O'Fallon, J. Hagemeister