Section 3.5
Overloading

1. Function overloading
2. Operator overloading

3.5.1 Function Overloading

- What is overloading?
  - it’s giving something multiple meanings, or multiple definitions
- Overloaded functions
  - have the same name
  - have different parameter types and/or diff. parameter ordering
  - can be global functions or member functions

Function Overloading (cont.)

- Characteristics
  - they must have a unique *signature* (not just a unique prototype!)
  - a unique return type is insufficient
- Convention
  - they should be used for *functionally related* tasks

Function Overloading (cont.)

- How it works
  - the compiler **mangles** every function name
    - it changes the function name to a combination of:
      - the function name
      - the ordered parameter types
    - this makes each function name unique
  - compiler chooses which function to call based on how it's called
    - the parameter types and their order
  - **`coding example <p1>`**

3.5.2 Operator Overloading

- Purpose
- Cascading
- Operators as functions
- Overloading:
  - stream insertion and extraction operators
  - unary and binary operators
  - operators on collection classes
  - increment and decrement operators

Purpose

- What is operator overloading?
  - it defines how operators work on *user-defined data types*
    - user-defined data types are our *classes*
  - example:

```
Student matilda, joe;
if (matilda > joe)
  cout << "Matilda wins!" << endl;
else
  cout << "Joe wins!" << endl;
```

  - what does the **>** operator do? what does the comparison mean?
  - **coding example <p2>**

Purpose (cont.)

- Why overload operators?
  - language consistency
  - code readability
  - because it’s cool !!

Purpose (cont.)

- Example: the C++ library class **`string`** provides:
  - assignment operator
  - relational operators
  - subscript operator
  - stream insertion and extraction operators
  - ... and lots more ...

Implicit and Explicit Overload

- Implicitly overloaded operators
  - assignment: `=`
  - address-of: `&`
  - sequencing: `,`

- Explicitly overloaded operators
  - the class developer decides
  - almost all operators can be overloaded
  - each operator must be overloaded *separately*
    - no freebies
    - if you implement + and =, you don't get += automatically

Dynamically Allocation

- Objects with dynamically allocated members are special
- You should provide:
  - a copy constructor
  - a destructor
  - an overloaded assignment operator

Approach to Overloading

- How are operators overloaded?
  - an operator is a *function*
    - the keyword **`operator`** followed by the operator symbol
  - the operands are the *parameters*
  - think about the **return type**!
    - how does it work for integers?
    - that's how it should work for your class
    - always enable *cascading* if that's how it works for ints

Approach to Overloading (cont.)

- Complication
  - operators can be overloaded as:
    - a global function
    - a member function
    - but not both
  - ... more on this later ...

Approach to Overloading (cont.)

- Limitations:
  - we cannot change operators for built-in (primitive) data types
  - we cannot create new operators
  - we cannot change an operator's arity
  - we cannot change an operator's precedence or associativity
  - we cannot overload non-overloadable operators
    - dot, scope resolution, conditional, and a few more

Cascading

- You must remember **`this`**
  - it's an object's **pointer** to itself
  - it is passed as an implicit parameter to all member functions
    - except static member functions
  - this can be used implicitly or explicitly

Cascading (cont.)

- What is cascading?
  - chaining together member function calls in a single statement
- How does it work?
  - a member function returns the object
    - how do you use **`this`** ?
  - the next member function operates on returned object
  - ... and so on ...
  - **`coding example <p3>`**

Operators as Functions

- Operators can be overloaded as either:
  - global functions
  - member functions
- Restriction
  - we cannot overload the same operator as both
    - the compiler considers them equivalent

Operators as Functions (cont.)

- Overloading operator as a member function
  - left-most operand is the *target* object (the **`this`** object)
    - the object on which the member function is called
    - dereference **`this`** pointer to access target object
  - remaining operand passed in as reference parameter
    - for binary operators

Operators as Functions (cont.)

- Overloading operator as global function
  - all operands are parameters
    - including target object (the **`this`** object)
  - function must be a friend to access private or protected members
  - **`coding example <p4>`**

Operators as Functions (cont.)

- Some operators **must** be overloaded as member functions
  - cast: `()`
  - subscript: `[]`
  - arrow: `->`

- Remember: always enable cascading where appropriate

Operators as Functions (cont.)

- Some operators **must** be overloaded as global functions
  - stream insertion: **<<**
  - stream extraction: **>>**
  - other operators: to enable commutativity

- Remember: always enable cascading where appropriate

Stream Operators

- Stream insertion and extraction operators
  - they are already overloaded for built-in types
  - they **must** be overloaded as global functions
    - why?
  - think about the return type
  - **`coding example <p5>`**

Stream Operators (cont.)

- Stream insertion operator << takes two operands
  - the left-hand side operand is **`cout`**
    - an object of **`ostream`** class
    - used as a reference
  - the right-hand side operand is the object to be output
  - it must be a friend function of the class to access class members

Stream Operators (cont.)

- Stream extraction operator **`>>`** takes two operands
  - the left-hand side operand is **`cin`**
    - an object of **`istream`** class
    - used as a reference
  - the right-hand side operand is the object to be input
  - it must be a friend function of the class to access class members

Unary Operators

- Overloaded as a global function
  - takes one parameter
    - object or reference to the **`this`** object (object operated on)
- Overloaded as a member function
  - takes no parameters
  - cannot be static

Binary Operators

- Overloaded as a global function
  - takes two parameters
    - first one must be an object or reference to the **`this`** object
- Overloaded as a member function
  - takes one parameter
  - cannot be static
- **`coding example <p6>`**

Operators on Collection Classes

- Many useful operators can be implemented
  - for accessing elements:
    - `[]`
  - for adding and removing elements
    - `+  +=`
    - `-  -=`
  - for comparing collections
    - `==  !=`
  - lots more!
  - **coding example <p7>**

Increment and Decrement Operators

- Many flavours
  - increment or decrement
  - prefix or postfix
  - global or member function
- Each has its own syntax

Increment and Decrement (cont.)

- Prefix ++ or --
  - modifies the target object
  - returns a reference to the object
  - as a member function
    - takes no parameters
  - as a global function
    - takes one parameter: a reference to the target object
  - **coding example <p8>**

Increment and Decrement (cont.)

- Complications
  - how do we distinguish prefix and postfix function prototypes?
  - prefix and postfix have different functionality
- Solution
  - a dummy parameter is introduced
  - we have to be smart about coding the postfix function

Increment and Decrement (cont.)

- Postfix ++ or --
  - makes a local copy of the target object
  - modifies the original object
  - returns the *local copy* of the object
    - **not** a reference!
    - you need a copy constructor
- Creation of temporary object makes postfix slower
  - always use prefix instead, if you can

Increment and Decrement (cont.)

- Postfix ++ or −− (cont.)
  - as a member function
    - takes two parameters:
      - a reference to the target object
      - a dummy integer
  - as a global function
    - takes one parameter: a dummy integer