Section 3.6
Templates

1. Overview
2. Function templates
3. Class templates
4. STL `vector` class

3.6.1 Overview

- Main principles of object-oriented design:
  - data abstraction
    - separation of *abstract* properties and *concrete* implementation
  - code reuse
    - reuse existing code
    - make your classes reusable

Overview (cont.)

- Important mechanisms for code reuse
  - use existing libraries
  - generic programming
    - templates in C++

Overview (cont.)

- Programming with templates
  - goal
    - code once, reuse many times
    - write code that works with all kinds of objects
  - approach
    - use a data type as a “parameter” to the template
      - templates can be functions or classes
    - the template code can be used with any data type

3.6.2 Function Templates

- Goal
  - create a function that works with any data type
- Characteristics
  - any function can be “templated” (made into a template)
    - global function
    - member function
  - if a member function is templated, its class must be too
    - ... more on this later ...
  - **`coding example <p1>`**

Function Templates (cont.)

- How it works:
  - what you do:
    - write the function using a “variable” in place of a data type
    - call the function using any data type
  - what the compiler does:
    - creates a **specialization** of the template for each data type **used**
      - a real function is placed in your code, with the data type hard-coded
      - specializations are only generated for *the data types you use* in the code
    - the new function is compiled with the rest of your code

Function Templates (cont.)

- Overloading templated with non-templated functions
  - templated function can be overloaded
    - same name, but with different parameters
    - this is not the same as a specialization
  - non-templated function will be chosen over templated one
  - **`coding example <p2>`**

3.6.3 Class Templates

- Goal
  - create a class that works with any data type
  - this class is known as a *parameterized type*

- Characteristics
  - any member of the class can use the parameter data type
    - data members
    - member functions
  - any member function can be templated
  - often used for collection classes

- **coding example <p3>**

Class Templates (cont.)

- How it works:
  - what you do:
    - write the class using a “variable” in place of a data type
    - create an instance of the class using any data type
  - what the compiler does:
    - creates a **specialization** of the template for each data type used
      - a real class is placed in your code, with the data type hard-coded
      - all templated members of the class are specialized for that type
      - specializations are only generated for *the data types you use* in the code
    - the new class is compiled with the rest of your code

Class Templates (cont.)

- Special cases:
  - multiple types
    - **coding example <p4>**
  - non-type parameter
    - **coding example <p5>**
  - default types
    - **coding example <p6>**
- Static members
  - each specialization gets its own copy of static members

3.6.4 STL `vector` Class

- Standard Template Library (STL)
  - provides many container classes and algorithms
  - ... more on this later ...

- What's wrong with primitive arrays?
  - well, they're primitive
  - they don't do bounds checking
  - they don't support any operators
    - assignment
    - relational
    - ... the list goes on ...

STL `vector` Class (cont.)

- One alternative: **`vector`** class template
  - it can store any data type
  - it supports many operators
  - it provides useful member functions
    - **`size()`**
    - **`at()`**
    - ... and many more ...
  - online documentation: www.cplusplus.com
  - **`coding example <p7>`**