Class/Function Templates

C++ How to Program, 9/e

Sources

- C++ How to Program, 9/e (by Paul Deitel and Harvey Deitel)
  - Chapter 18:
    Introduction to Custom Templates

Outline

- Introduction (Section 18.1)
- Class Templates (Section 18.2)
- Function Template to Manipulate a Class-Template Specialization Object (Section 18.3)

Outline

- **Introduction (Section 18.1)**
- Class Templates (Section 18.2)
- Function Template to Manipulate a Class-Template Specialization Object (Section 18.3)

Generic programming

- Generic programming is a style of computer programming
  - in which algorithms are written in terms of *types to-be-specified-later* that are then *instantiated* when needed for specific types provided as *parameters*;
  - permitting writing common *functions* or *types* that differ only in the set of types on which they operate when used, thus reducing duplication.

Generic programming (cont.)

- C++ supports generic programming in terms of:
  - class templates: enable you to conveniently specify a variety of related classes—called class-template specializations.
  - function templates: enable you to conveniently specify a variety of related (overloaded) functions—called function-template specializations.

Outline

- Introduction (Section 18.1)
- **Class Templates (Section 18.2)**
- Function Template to Manipulate a Class-Template Specialization Object (Section 18.3)

Class Templates

- Consider stack:
  - a data structure into which we insert items *only* at the *top* and retrieve those items *only* from the *top* in *last-in*, *first-out order*.
- We find:
  - The concept of a stack is *independent of the type of the items* being placed in the stack.
  - However, to *instantiate* a stack, a data type must be specified.
- Thus, we define a stack *generically* then use *type-specific* versions of this generic stack class.

Class Templates (cont.)

- Class templates are called parameterized types.
  - E.g., ***Class Template*** ***`Stack< T >`***
  - They require one or more *type parameters*, *i.e.*, ***`T`*** *in* ***`Stack< T >`***, to specify how to customize a generic class template to form a ***class-template specialization***.
    - E.g., a particular *class-template specialization* ***`Stack< double >`*** (the compiler writes the specialization source code).

Creating Class Template Stack< T >

- The `Stack` class-template definition in Fig. 18.1 looks like a conventional class definition, with a few key differences.
  - It's preceded by line 7

```
template< typename T >
```

  - All class templates begin with keyword `template` followed by a list of template parameters enclosed in angle brackets (`<` and `>`).
  - Each template parameter that represents a type *must* be preceded by either of the interchangeable keywords `typename` or `class`.

```
// Fig. 18.1: Stack.h
// Stack class template.
#ifndef STACK_H
#define STACK_H
#include <deque>

template< typename T >
class Stack
{
public:
    // return the top element of the Stack
    T& top()
    {
        return stack.front();
    } // end function template top

    // push an element onto the Stack
    void push( const T &pushValue )
    {
        stack.push_front( pushValue );
    } // end function template push

```

Fig. 18.1 | Stack class template. (Part 1 of 2.)

```
   // pop an element from the stack
   void pop()
   {
      stack.pop_front();
   } // end function template pop

   // determine whether Stack is empty
   bool isEmpty() const
   {
      return stack.empty();
   } // end function template isEmpty

   // return size of Stack
   size_t size() const
   {
      return stack.size();
   } // end function template size

private:
   std::deque< T > stack; // internal representation of the Stack
}; // end class template Stack

#endif
```

Fig. 18.1 | Stack class template. (Part 2 of 2.)

Creating Class Template Stack< T > (cont.)

- The type parameter `T` acts as a placeholder for the `Stack`'s element type.
  - The element type is mentioned generically throughout the `Stack` class-template definition as `T` (lines 12, 18 and 42).
- The type parameter `T` becomes associated with a specific type (e.g., ***`int`*** or ***`double`***) when you create an object using the class template.
  - At that point, the compiler generates a copy of the class template in which all occurrences of the type parameter are replaced with the specified type.
- **NOTE**: we did not separate the class template's interface from its implementation.

Software Engineering Observation 18.2

Templates are typically defined in headers, which are then `#included` in the appropriate client source-code files. For class templates, this means that the member functions are also defined in the header—typically inside the class definition's body, as we do in Fig. 18.1.

Class Template Stack<T>'s Data Representation

- A `Stack` requires insertions and deletions *only* at its *top*.
- So, for example, the C++ Standard Library's `deque` (`double-ended queue`) could be used to store the `Stack`'s elements.
  - A `deque` supports fast insertions and deletions at its *front* and its *back*.
    - A `deque` is typically implemented as list of fixed-size, built-in arrays—new fixed-size built-in arrays are added as necessary.
  - Actually, a `deque` is the default representation for the Standard Library's `stack` adapter.

Class Template Stack<T>'s Member Functions

- The member-function definitions of a class template are *function templates*.
  - They use the class template's template parameter `T` to represent the element type.
- Our `Stack` class template does not define it's own constructors.
  - The *default constructor* provided by the compiler will invoke the deque's default constructor.

Class Template Stack<T>'s Member Functions (cont.)

- We also provide the following member functions in Fig. 18.1:
  - `top` (lines 12–15) returns a reference to the `Stack`'s top element.
  - `push` (lines 18–21) places a new element on the top of the `Stack`.
  - `pop` (lines 24–27) removes the `Stack`'s top element.
  - `isEmpty` (lines 30–33) returns a `bool` value—`true` if the `Stack` is empty and `false` otherwise.
  - `size` (lines 36–39) returns the number if elements in the `Stack`.
- Each of these member functions *delegates* its responsibility to the appropriate member function of class template `deque`.

Declaring a Class Template's Member Functions Outside the Class Template Definition

- Though we did *not* do so in our `Stack` class template, member-function definitions can appear *outside* a class template definition.
- If you do this, each must begin with the `template` keyword followed by the *same* set of template parameters as the class template.
- In addition, the member functions must be qualified with the class name and scope resolution operator.

Declaring a Class Template's Member Functions Outside the Class Template Definition (cont.)

- For example, you can define the `pop` function outside the class-template definition as follows:

```
template< typename T >
inline void Stack<T>::pop()
{
   stack.pop_front();
} // end function template pop
```

- `Stack<T>::` indicates that pop is in the scope of class `Stack<T>`.
- The Standard Library's container classes tend to define all their member functions *inside* their class definitions.

Testing Class Template Stack<T>

- Now, let's consider the driver that exercises the `Stack` class template in Fig. 18.2.
- The driver begins by instantiating object `doubleStack` (line 9).
  - This object is declared as a `Stack< double >` (pronounced "`Stack` of `double`").
- The driver then instantiates `int` stack `intStack` with the declaration (line 34):
  - `Stack< int > intStack;` (pronounced "`intStack` is a `Stack` of `int`").

```cpp
// Fig. 18.2: fig18_02.cpp
// Stack class template test program.
#include <iostream>
#include "Stack.h" // Stack class template definition
using namespace std;

int main()
{
   Stack< double > doubleStack; // create a Stack of double
   const size_t doubleStackSize = 5; // stack size
   double doubleValue = 1.1; // first value to push

   cout << "Pushing elements onto doubleStack\n";

   // push 5 doubles onto doubleStack
   for ( size_t i = 0; i < doubleStackSize; ++i )
   {
      doubleStack.push( doubleValue );
      cout << doubleValue << ' ';
      doubleValue += 1.1;
   } // end while

   cout << "\n\nPopping elements from doubleStack\n";

```

Fig. 18.2 | Stack class template test program. (Part 1 of 3.)

```
   // pop elements from doubleStack
   while ( !doubleStack.isEmpty() ) // loop while Stack is not empty
   {
      cout << doubleStack.top() << ' '; // display top element
      doubleStack.pop(); // remove top element
   } // end while

   cout << "\nStack is empty, cannot pop.\n";

   Stack< int > intStack; // create a Stack of int
   const size_t intStackSize = 10; // stack size
   int intValue = 1; // first value to push

   cout << "\nPushing elements onto intStack\n";

   // push 10 integers onto intStack
   for ( size_t i = 0; i < intStackSize; ++i )
   {
      intStack.push( intValue );
      cout << intValue++ << ' ';
   } // end while

   cout << "\n\nPopping elements from intStack\n";

```

Fig. 18.2 | Stack class template test program. (Part 2 of 3.)

```cpp
    // pop elements from intStack
    while ( !intStack.isEmpty() ) // loop while Stack is not empty
    {
       cout << intStack.top() << ' '; // display top element
       intStack.pop(); // remove top element
    } // end while

    cout << "\nStack is empty, cannot pop." << endl;
} // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty, cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty, cannot pop
```

Fig. 18.2 | Stack class template test program. (Part 3 of 3.)

Outline

- Introduction (Section 18.1)
- Class Templates (Section 18.2)
- **Function Template to Manipulate a Class-Template Specialization Object (Section 18.3)**

Function Template to Manipulate a Class-Template Specialization Object

- Notice that the code in function `main` of Fig. 18.2 is *almost identical* for both
  - the `doubleStack` manipulations in lines 9–32 and
  - the `intStack` manipulations in lines 34–56.
- This presents another opportunity to use a function template `testStack` (lines 10–39) in Figure 18.3, performing the same tasks as `main` in Fig. 18.2:
  1. push a series of values onto a `Stack<T>` and
  2. pop the values off a `Stack<T>`.

```cpp
// Fig. 18.3: fig18_03.cpp
// Passing a Stack template object
// to a function template.
#include <iostream>
#include <string>
#include "Stack.h" // Stack class template definition
using namespace std;

// function template to manipulate Stack< T >
template< typename T >
void testStack(
   Stack< T > &theStack, // reference to Stack< T >
   const T &value, // initial value to push
   const T &increment, // increment for subsequent values
   size_t size, // number of items to push
   const string &stackName ) // name of the Stack< T > object
{
   cout << "\nPushing elements onto " << stackName << '\n';
   T pushValue = value;

```

Fig. 18.3 | Passing a Stack template object to a function template. (Part 1 of 4.)

```cpp
    // push element onto Stack
    for ( size_t i = 0; i < size; ++i )
    {
       theStack.push( pushValue ); // push element onto Stack
       cout << pushValue << ' ';
       pushValue += increment;
    } // end while

    cout << "\n\nPopping elements from " << stackName << '\n';

    // pop elements from Stack
    while ( !theStack.isEmpty() ) // loop while Stack is not empty
    {
       cout << theStack.top() << ' ';
       theStack.pop(); // remove top element
    } // end while

```

Fig. 18.3 | Passing a `Stack` template object to a function template. (Part 2 of 4.)

```
    cout << "\nStack is empty. Cannot pop." << endl;
} // end function template testStack

int main()
{
    Stack< double > doubleStack;
    const size_t doubleStackSize = 5;
    testStack( doubleStack, 1.1, 1.1, doubleStackSize, "doubleStack" );

    Stack< int > intStack;
    const size_t intStackSize = 10;
    testStack( intStack, 1, 1, intStackSize, "intStack" );
} // end main
```

Fig. 18.3 | Passing a `Stack` template object to a function template. (Part 3 of 4.)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty, cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty, cannot pop
```

Fig. 18.3 | Passing a `Stack` template object to a function template. (Part 4 of 4.)

Function Template to Manipulate a Class-Template Specialization Object (cont.)

- Function template `testStack` uses `T` (specified at line 10) to represent the data type stored in the `Stack<T>`.
- The function template takes five arguments (lines 12–16):
  1. the `Stack<T>` to manipulate
  2. a value of type `T` that will be the first value `push`ed onto the `Stack<T>`
  3. a value of type `T` used to increment the values `push`ed onto the `Stack<T>`
  4. the number of elements to push onto the `Stack<T>`
  5. a `string` that represents the name of the `Stack<T>` object for output purposes

Function Template to Manipulate a Class-Template Specialization Object (cont.)

- Function `main` (lines 41–50) instantiates
  - an object of type `Stack<double>` called `doubleStack` (line 43)
  - an object of type `Stack<int>` called `intStack` (line 47)

  and uses these objects in lines 45 and 49.
- The compiler infers the type of `T` for `testStack` from the type used to instantiate the function's first argument
  - i.e., the type used to instantiate `doubleStack` or `intStack`.