



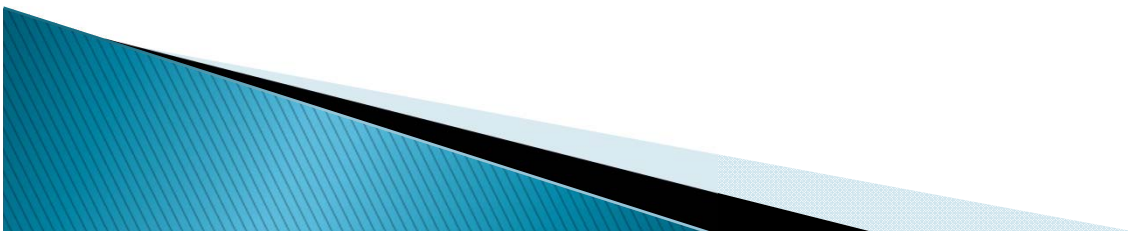
# Class/Function Templates

C++ How to Program, 9/e



# Sources

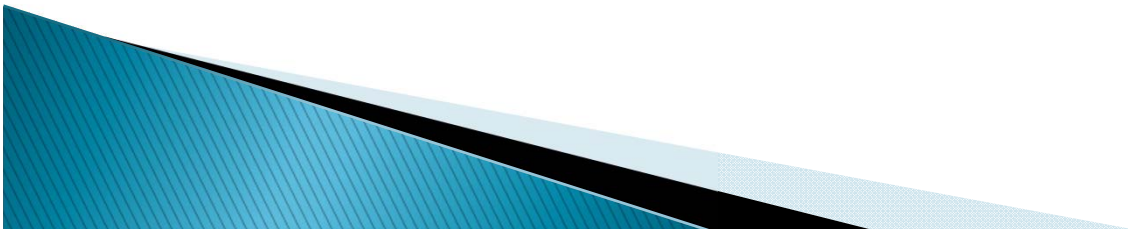
- ▶ C++ How to Program, 9/e (by Paul Deitel and Harvey Deitel)
  - Chapter 18:  
Introduction to Custom Templates





# 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*.



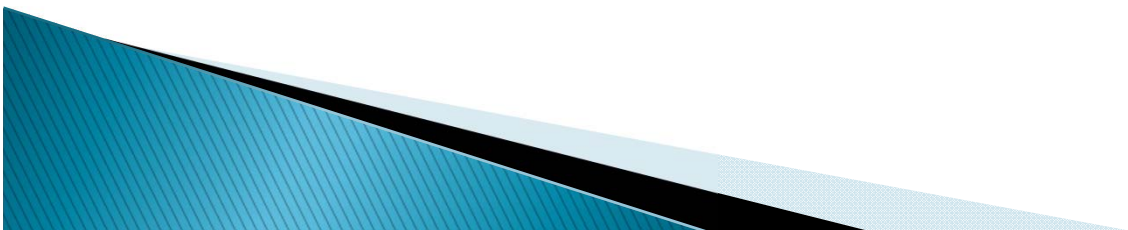
# Generic programming (cont.)

- ▶ 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`.



---

```
1 // Fig. 18.1: Stack.h
2 // Stack class template.
3 #ifndef STACK_H
4 #define STACK_H
5 #include <deque>
6
7 template< typename T >
8 class Stack
9 {
10 public:
11     // return the top element of the Stack
12     T& top()
13     {
14         return stack.front();
15     } // end function template top
16
17     // push an element onto the Stack
18     void push( const T &pushValue )
19     {
20         stack.push_front( pushValue );
21     } // end function template push
22
```

---

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



```
23    // pop an element from the stack
24    void pop()
25    {
26        stack.pop_front();
27    } // end function template pop
28
29    // determine whether Stack is empty
30    bool isEmpty() const
31    {
32        return stack.empty();
33    } // end function template isEmpty
34
35    // return size of Stack
36    size_t size() const
37    {
38        return stack.size();
39    } // end function template size
40
41    private:
42        std::deque< T > stack; // internal representation of the Stack
43    }; // end class template Stack
44
45    #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` (**d**ouble-**e**nded **q**ueue) 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 its 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 of 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 `doubl eStack` (line 9).
  - This object is declared as a `Stack< doubl e >` (pronounced “Stack of doubl e”).
- ▶ The driver then instantiates `i nt stack i ntStack` with the declaration (line 34):
  - `Stack< i nt > i ntStack;` (pronounced “i ntStack is a Stack of i nt”).



```
1 // Fig. 18.2: fig18_02.cpp
2 // Stack class template test program.
3 #include <iostream>
4 #include "Stack.h" // Stack class template definition
5 using namespace std;
6
7 int main()
8 {
9     Stack< double > doubleStack; // create a Stack of double
10    const size_t doubleStackSize = 5; // stack size
11    double doubleValue = 1.1; // first value to push
12
13    cout << "Pushing elements onto doubleStack\n";
14
15    // push 5 doubles onto doubleStack
16    for ( size_t i = 0; i < doubleStackSize; ++i )
17    {
18        doubleStack.push( doubleValue );
19        cout << doubleValue << ' ';
20        doubleValue += 1.1;
21    } // end while
22
23    cout << "\n\nPopping elements from doubleStack\n";
24
```

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



```
25 // pop elements from doubleStack
26 while ( !doubleStack.isEmpty() ) // loop while Stack is not empty
27 {
28     cout << doubleStack.top() << ' '; // display top element
29     doubleStack.pop(); // remove top element
30 } // end while
31
32 cout << "\nStack is empty, cannot pop.\n";
33
34 Stack< int > intStack; // create a Stack of int
35 const size_t intStackSize = 10; // stack size
36 int intValue = 1; // first value to push
37
38 cout << "\nPushing elements onto intStack\n";
39
40 // push 10 integers onto intStack
41 for ( size_t i = 0; i < intStackSize; ++i )
42 {
43     intStack.push( intValue );
44     cout << intValue++ << ' ';
45 } // end while
46
47 cout << "\n\nPopping elements from intStack\n";
48
```

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





```
49 // pop elements from intStack
50 while ( !intStack.isEmpty() ) // loop while Stack is not empty
51 {
52     cout << intStack.top() << ' '; // display top element
53     intStack.pop(); // remove top element
54 } // end while
55
56 cout << "\nStack is empty, cannot pop." << endl;
57 } // 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>`.



```
1 // Fig. 18.3: fig18_03.cpp
2 // Passing a Stack template object
3 // to a function template.
4 #include <iostream>
5 #include <string>
6 #include "Stack.h" // Stack class template definition
7 using namespace std;
8
9 // function template to manipulate Stack< T >
10 template< typename T >
11 void testStack(
12     Stack< T > &theStack, // reference to Stack< T >
13     const T &value, // initial value to push
14     const T &increment, // increment for subsequent values
15     size_t size, // number of items to push
16     const string &stackName ) // name of the Stack< T > object
17 {
18     cout << "\nPushing elements onto " << stackName << '\n';
19     T pushValue = value;
20
```

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



---

```
21    // push element onto Stack
22    for ( size_t i = 0; i < size; ++i )
23    {
24        theStack.push( pushValue ); // push element onto Stack
25        cout << pushValue << ' ';
26        pushValue += increment;
27    } // end while
28
29    cout << "\n\nPopping elements from " << stackName << '\n';
30
31    // pop elements from Stack
32    while ( !theStack.isEmpty() ) // loop while Stack is not empty
33    {
34        cout << theStack.top() << ' ';
35        theStack.pop(); // remove top element
36    } // end while
37
```

---

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



---

```
38     cout << "\nStack is empty. Cannot pop." << endl;
39 } // end function template testStack
40
41 int main()
42 {
43     Stack< double > doubleStack;
44     const size_t doubleStackSize = 5;
45     testStack( doubleStack, 1.1, 1.1, doubleStackSize, "doubleStack" );
46
47     Stack< int > intStack;
48     const size_t intStackSize = 10;
49     testStack( intStack, 1, 1, intStackSize, "intStack" );
50 } // 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 pushed onto the `Stack<T>`
  3. a value of type `T` used to increment the values pushed 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`.