14

Templates

Behind that outside pattern
the dim shapes get clearer every day.
It is always the same shape, only very numerous.

—Charlotte Perkins Gilman

Every man of genius sees the world at a different angle from his fellows.

—Havelock Ellis

...our special individuality, as distinguished from our generic humanity.

—Oliver Wendell Holmes, Sr

OBJECTIVES

In this chapter you will learn:

- To use function templates to conveniently create a group of related (overloaded) functions.
- To distinguish between function templates and function-template specializations.
- To use class templates to create a group of related types.
- To distinguish between class templates and class-template specializations.
- To overload function templates.
- To understand the relationships among templates, friends, inheritance and static members.

Outline

14.1 Introduction

- **Function templates and class templates**
  - **Enable programmers to specify an entire range of related functions and related classes**
    - **Called function-template specializations and class-template specializations, respectively**
  - **Generic programming**
  - **Analogy: templates are like stencils, template specializations are like separate tracings**
    - **All tracings have the same shape, but they could have different colors**

Software Engineering Observation 14.1

Most C++ compilers require the complete definition of a template to appear in the client source-code file that uses the template. For this reason and for reusability, templates are often defined in header files, which are then `#included` into the appropriate client source-code files. For class templates, this means that the member functions are also defined in the header file.

14.2 Function Templates

- Function Templates
  - Used to produce overloaded functions that perform identical operations on different types of data
    - Programmer writes a single function-template definition
    - Compiler generates separate object-code functions (function-template specializations) based on argument types in calls to the function template
  - Similar to macros in C, but with full type checking

Error-Prevention Tip 14.1

Function templates, like macros, enable software reuse. Unlike macros, function templates help eliminate many types of errors through the scrutiny of full C++ type checking.

14.2 Function Templates (Cont.)

- **Function-template definitions**
  - Preceded by a template header
    - Keyword `template`
    - List of template parameters
      - Enclosed in angle brackets (< and >)
      - Each template parameter is preceded by keyword `class` or keyword `typename` (both are interchangeable)
      - Used to specify types of arguments to, local variables in and return type of the function template
    - Examples
      - `template< typename T >`
      - `template< class ElementType >`
      - `template< typename BorderType, typename Filltype >`

Common Programming Error 14.1

Not placing keyword `class` or keyword `typename` before each type template parameter of a function template is a syntax error.

Outline

Type template parameter **T** specified in template header

fig14_01.cpp

(1 of 2)

```cpp
// Fig. 14.1: fig14_01.cpp
// Using template functions.
#include <iostream>
using std::cout;
using std::endl;

// function template printArray definition
template< typename T >
void printArray( const T *array, int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
} // end function template printArray

int main()
{
   const int ACOUNT = 5; // size of array a
   const int BCOUNT = 7; // size of array b
   const int CCOUNT = 6; // size of array c

   int a[ ACOUNT ] = { 1, 2, 3, 4, 5 };
   double b[ BCOUNT ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ CCOUNT ] = "HELLO"; // 6th position for null

   cout << "Array a contains:" << endl;
```

```

   // call integer function-template specialization
   printArray( a, ACOUNT );

   cout << "Array b contains:" << endl;

   // call double function-template specialization
   printArray( b, BCOUNT );

   cout << "Array c contains:" << endl;

   // call character function-template specialization
   printArray( c, CCOUNT );
   return 0;
} // end main
```

Creates a function-template specialization of **printArray** where **int** replaces **T**

Creates a function-template specialization of **printArray** where **double** replaces **T**

Creates a function-template specialization of **printArray** where **char** replaces **T**

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Common Programming Error 14.2

If a template is invoked with a user-defined type, and if that template uses functions or operators (e.g., ==, +, <=) with objects of that class type, then those functions and operators must be overloaded for the user-defined type. Forgetting to overload such operators causes compilation errors.

Performance Tip 14.1

Although templates offer software-reusability benefits, remember that multiple function-template specializations and class-template specializations are instantiated in a program (at compile time), despite the fact that the template is written only once. These copies can consume considerable memory. This is not normally an issue, though, because the code generated by the template is the same size as the code the programmer would have written to produce the separate overloaded functions.

14.3 Overloading Function Templates

- A function template may be overloaded
  - Other function templates that specify the same name but different parameters
  - Nontemplate functions that specify the same name but different parameters
  - The compiler chooses the best function or specialization to match the function call
    - A nontemplate function is chosen over a template specialization in case of a tie
    - Otherwise, multiple matches results in a compilation error (the compiler considers the function call to be an ambiguous function call)

Common Programming Error 14.3

If no matching function definition can be found for a particular function call, or if there are multiple matches, the compiler generates an error.

14.4 Class Templates

- Class templates (or parameterized types)
  - Class-template definitions are preceded by a header
    - Such as `template< typename T >`
  - Type parameter `T` can be used as a data type in member functions and data members
  - Additional type parameters can be specified using a comma-separated list
    - As in `template< typename T1, typename T2 >`

Software Engineering Observation 14.2

Class templates encourage software reusability by enabling type-specific versions of generic classes to be instantiated.

```
// Fig. 14.2: Stack.h
// Stack class template.
#ifndef STACK_H
#define STACK_H

template< typename T >
class Stack
{
public:
   Stack( int = 10 ); // default constructor (Stack size 10)

   // destructor
   ~Stack()
   {
      delete [] stackPtr; // deallocate internal space for Stack
   } // end ~Stack destructor

   bool push( const T& ); // push an element onto the Stack
   bool pop( T& ); // pop an element off the Stack

   // determine whether Stack is empty
   bool isEmpty() const
   {
      return top == -1;
   } // end function isEmpty
```

Create class template **`Stack`** with type parameter **`T`**

Member functions that use type parameter **`T`** in specifying function parameters

`Stack.h`

Outline

```
// determine whether Stack is full
bool isFull() const
{
   return top == size - 1;
} // end function isFull

private:
   int size; // # of elements in the Stack
   int top; // location of the top element (-1 means empty)
   T *stackPtr; // pointer to internal representation of the Stack
}; // end class template Stack

// constructor template
template< typename T >
Stack< T >::Stack( int s )
   : size( s > 0 ? s : 10 ), // validate size
     top( -1 ), // Stack initially empty
     stackPtr( new T[ size ] ) // allocate memory for elements
{
   // empty body
} // end Stack constructor template
```

`Stack.h`

(2 of 3)

Data member **`stackPtr`** is a pointer to a **`T`**

Member-function template definitions that appear outside the class-template definition begin with the template header

Outline

Stack.h

(3 of 3)

```
// push element onto Stack;
// if successful, return true; otherwise, return false
template< typename T >
bool Stack< T >::push( const T &pushValue )
{
   if ( !isFull() )
   {
      stackPtr[ ++top ] = pushValue; // place item on Stack
      return true; // push successful
   } // end if

   return false; // push unsuccessful
} // end function template push

// pop element off Stack;
// if successful, return true; otherwise, return false
template< typename T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() )
   {
      popValue = stackPtr[ top-- ]; // remove item from Stack
      return true; // pop successful
   } // end if

   return false; // pop unsuccessful
} // end function template pop

#endif
```

```
// Fig. 14.3: fig14_03.cpp
// Stack class template test program.
#include <iostream>
using std::cout;
using std::endl;

#include "Stack.h" // Stack class template definition

int main()
{
   Stack< double > doubleStack( 5 ); // size 5
   double doubleValue = 1.1;

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

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

   cout << "\nStack is full. Cannot push " << doubleValue
      << "\n\nPopping elements from doubleStack\n";

   // pop elements from doubleStack
   while ( doubleStack.pop( doubleValue ) )
      cout << doubleValue << ' ';
```

Create class-template specialization **`Stack< double >`** where type **`double`** is associated with type parameter **`T`** fig14_03.cpp

```

   cout << "\nStack is empty. Cannot pop\n";

   Stack< int > intStack; // default size 10
   int intValue = 1;
   cout << "\nPushing elements onto intStack\n";

   // push 10 integers onto intStack
   while ( intStack.push( intValue ) )
   {
      cout << intValue << ' ';
      intValue++;
   } // end while

   cout << "\nStack is full. Cannot push " << intValue
      << "\n\nPopping elements from intStack\n";

   // pop elements from intStack
   while ( intStack.pop( intValue ) )
      cout << intValue << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
   return 0;
} // end main
```

Create class-template specialization **`Stack< int >`** where type **`int`** is associated with type parameter **`T`**

Outline

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

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
Stack is full. Cannot push 11

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

fig14_03.cpp

(3 of 3)

```cpp
// Fig. 14.4: fig14_04.cpp
// Stack class template test program. Function main uses a
// function template to manipulate objects of type Stack< T >.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

#include "Stack.h" // Stack class template definition

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

   // push element onto Stack
   while ( theStack.push( value ) )
   {
      cout << value << ' ';
      value += increment;
   } // end while
```

fig14_04.cpp

Use a function template to process **Stack** class-template specializations

Outline

fig14_04.cpp

(2 of 3)

```
   cout << "\nStack is full. Cannot push " << value
      << "\n\nPopping elements from " << stackName << '\n';

   // pop elements from Stack
   while ( theStack.pop( value ) )
      cout << value << ' ';

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

int main()
{
   Stack< double > doubleStack( 5 ); // size 5
   Stack< int > intStack; // default size 10

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );

   return 0;
} // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

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
Stack is full. Cannot push 11

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

fig14_04.cpp

(3 of 3)

14.5 Nontype Parameters and Default Types for Class Templates

- Nontype template parameters
  - Can have default arguments
  - Are treated as `const`s
  - Example
    - Template header:

```
template< typename T, int elements >
```

      Declaration:

```
Stack< double, 100 > salesFigures;
```

- Type parameters can have default arguments too
  - Example
    - Template header:

```
template< typename T = string >
```

      Declaration:

```
Stack<> jobDescriptions;
```

Performance Tip 14.2

When appropriate, specify the size of a container class (such as an array class or a stack class) at compile time (possibly through a nontype template parameter). This eliminates the execution-time overhead of using `new` to create the space dynamically.

Software Engineering Observation 14.3

Specifying the size of a container at compile time avoids the potentially fatal execution-time error if `new` is unable to obtain the needed memory.

14.5 Nontype Parameters and Default Types for Class Templates (Cont.)

- **Explicit specializations**
  - Used when a particular type will not work with the general template or requires customized processing
  - Example for an explicit `Stack< Employee >` specialization
    - ```
      template<>
      class Stack< Employee >
      {
           …
      };
      ```
  - Are complete replacements for the general template
    - Do not use anything from the original class template and can even have different members
      ```

14.6 Notes on Templates and Inheritance

- Templates and inheritance
  - A class template can be derived from a class-template specialization
  - A class template can be derived from a nontemplate class
  - A class-template specialization can be derived from a class-template specialization
  - A nontemplate class can be derived from a class-template specialization

14.7 Notes on Templates and Friends

- **Templates and friends**
  - Assume class template `X` with type parameter `T` as in:
    `template< typename T > class X`
    - A function can be the friend of every class-template specialization instantiated from a class template
      - `friend void f1();`
        - `f1` is a friend of `X< double >`, `X< string >`, etc.
    - A function can be the friend of only a class-template specialization with the same type argument
      - `friend void f2( X< T > & );`
        - `f2( X< float > & )` is a friend of `X< float >` but not a friend of `X< string >`

14.7 Notes on Templates and Friends (Cont.)

- A member function of another class can be the friend of every class-template specialization instantiated from a class template
  - `friend void A::f3();`
    - `f3` of class `A` is a friend of `X< double >`, `X< string >`, etc.
- A member function of another class can be the friend of only a class-template specialization with the same type argument
  - `friend void C< T >::f4( X< T > & );`
    - `C< float >::f4( X< float > & )` is a friend of `X< float >` but not a friend of `X< string >`

14.7 Notes on Templates and Friends (Cont.)

- Another class can be the friend of every class-template specialization instantiated from a class template
  - `friend class Y;`
    - Every member function of class Y is a friend of `X< double >`, `X< string >`, etc.
- A class-template specialization can be the friend of only a class-template specialization with the same type parameter
  - `friend class Z< T >;`
    - Class-template specialization `Z< float >` is a friend of `X< float >`, `Z< string >` is a friend of `X< string >`, etc.

14.8 Notes on Templates and `static` Members

- `static` data members of a class template
  - Each class-template specialization has its own copy of each `static` data member
    - All objects of that specialization share that one `static` data member
    - `static` data members must be defined and, if necessary, initialized at file scope
  - Each class-template specialization gets its own copy of the class template's `static` member functions