LAB12: Template

Array Class Template

Rung-Bin Lin

International Program in Informatics for Bachelor
Yuan Ze University

5/20/2021

Templates

- Function templates and class templates enable you to specify, with a single code segment, an entire range of related (overloaded) functions—called function-template specializations—or an entire range of related classes—called class-template specializations.
- This technique is called generic programming.
- Distinctions between templates and template specializations
  - Function templates and class templates are like stencils out of which we trace shapes.
  - Function-template specializations and class-template specializations are like the separate tracings that all have the same shape, but could, for example, be drawn in different colors.

Stencil

Function Templates

- **Overloaded functions normally perform *similar or identical operations on different types of data.***
- **If the operations are *identical for each type, they can be expressed more compactly and conveniently using function templates.***
- **Initially, you write a single function-template definition. Based on the argument types provided explicitly or inferred from calls to this function, the compiler generates separate source-code functions (i.e., function-template specializations) to handle each function call appropriately**

Function Templates (cont.)

- All function-template definitions 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 `class` or `typename`, as in

```
        template<typename T>
Or
        template<class ElementType>
Or
        template<typename BorderType, typename FillType>
```

Type parameters

- The type parameters, such as T and ElementType, of a function-template definition are used to specify the types of the arguments to the function, to specify the return type of the function, and to declare variables within the function.
- Keywords `typename` and `class` used to specify function-template parameters actually mean "any fundamental type or user-defined type."

PrintArray Example (1)

```
void printArray( const int * const
array, int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

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

```
void printArray( const double * const
array, int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

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

```
void printArray( const char * const
array, int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

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

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

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

printArray Example (2)

T ← int, array ← x, count ← 5

```
void printArray( const int * const
array, int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

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

Function

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

// function template printArray definition
template< typename T >
void printArray( const T * const 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

```

// If we make the following call:
int x[]={2,4 ,6,8,10};
printArray(x, 5);

Fig. 14.1 | Function-template specializations of function template `printArray`. (Part 1 of 3.)

```cpp
   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 );
} // end main
```

T is replaced by **int**.

`// Create a function: void printArray( const int * const array, int count )`

T is replaced by **double**.

`// Create a function: void printArray( const double * const array, int count )`

T is replaced by **char**.

`// Create a function: void printArray( const char * const array, int count )`

Fig. 14.1 | Function-template specializations of function template `printArray`. (Part 2 of 3.)

```
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
```

Class Templates

- **It's possible to understand the concept of a "stack" (a data structure into which we insert items at the top and retrieve those items in last-in, first-out order) independent of the type of the items being placed in the stack.**
- **However, to instantiate a stack, a data type must be specified.**
- **We need the means for describing the notion of a stack generically and instantiating classes that are type-specific versions of this generic stack class.**
- **C++ provides this capability through class templates.**
- **Class templates are called parameterized types, because they require one or more type parameters to specify how to customize a "generic class" template to form a class-template specialization.**

Stack Class Template

```
class StackInt {
 public:
  StackInt( int = 10 ); // default constructor (Stack size 10)
  bool push( const int & ); // push an element onto the Stack
  bool pop( int & ); // pop an element off the Stack
…
private:
  int size; // # of elements in the stack
  int top; // location of the top element (-1 means empty)
  int *stackPtr; // pointer to the Stack
}; // end class template Stack
```

```
class StackDouble {
  public:
  StacDoublek( int = 10 ); // default constructor (Stack size 10)
  bool push( const double & ); // push an element to the Stack
  bool pop( double & ); // pop an element off the Stack
…
private:
  int size; // # of elements in the stack
  int top; // location of the top element (-1 means empty)
  double *stackPtr; // pointer the Stack
}; // end class template Stack
```

```
class StackChar {
  public:
  StackChar( int = 10 ); // default constructor (Stack size 10)
  bool push( const char & ); // push an element onto the Stack
  bool pop( char & ); // pop an element off the Stack
…
private:
  int size; // # of elements in the stack
  int top; // location of the top element (-1 means empty)
  char *stackPtr; // pointer the Stack
}; // end class template Stack
```

```
template< typename T >
class Stack {
public:
  Stack( int = 10 ); // default constructor (Stack size 10)
  bool push( const T & ); // push an element onto the Stack
  bool pop( T & ); // pop an element off the Stack
…
private:
  int size; // # of elements in the stack
  int top; // location of the top element (-1 means empty)
  T *stackPtr; // pointer to  the Stack
}; // end class template Stack
```

```
// Creating stacks of different types
Stack< int > intStack;
Stack< double > doubleStack(20);
Stack < char > charStack(120);
```

Stack Class Template

- **It looks like a conventional class definition, except that it's preceded by the header**

```
template< typename T >
```

to specify a class-template definition with type parameter `T` which acts as a placeholder for the type of the `Stack` class to be created.

```
// 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

   // 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

```

// Creating member functions

// Every member function must be preceded by
// the header template< typename T >

// A function name must be preceded by the scope
// specification Stack< T> :: .

```cpp
// 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

// 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
```

```cpp
// 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
```

```cpp
// Fig. 14.3: fig14_03.cpp
// Stack class template test program.
#include <iostream>
#include "Stack.h" // Stack class template definition
using namespace std;

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 << ' ';

   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++ << ' ';
   } // 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;
} // end main
```

Output from Main() function

```
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
```

Non-type Parameters and Default Types for Class Templates

- Class template `Stack` of Section 14.4 used only a type parameter in the template header (Fig. 14.2, line 6).
- It's also possible to use non-type template parameters, which can have default arguments and are treated as `const`s.
- For example, the template header could be modified to take an `int elements` parameter as follows:

```
// nontype parameter elements
template< typename T, int elements >
```

- Then, a declaration such as

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

could be used to instantiate (at compile time) a 100-element `Stack` class-template specialization of `double` values named `mostRecentSalesFigures`; this class-template specialization would be of type `Stack< double, 100 >`.

Non-type Parameters and Default Types for Class Templates (cont.)

- The class definition then might contain a `private` data member with an array declaration such as

```
// array to hold Stack contents
T stackHolder[ elements ];
```

- A type parameter can specify a default type.

  For example,

```
// defaults to type string
template< typename T = string >
```

  specifies that a `T` is `string` if not specified otherwise.

- Then, a declaration such as

```
Stack<> jobDescriptions;
```

  could be used to instantiate a `Stack` class-template specialization of `string`s named `job-Descriptions`; this class-template specialization would be of type `Stack< string >`.

- Default type parameters must be the rightmost (trailing) parameters in a template's type-parameter list.

Lab 12: Array Class Template

- **Modify the Array class in Fig. 11.6~Fig. 7 into a class template that can be used to create an array of any data type.**
- **Add a member function void swap(int, int) to interchange the element denoted by the first parameter with the element denoted by the second parameter. If any subscript is out of range, no swapping is done and program execution continues, but "** Error: swapping fails, subscript X is out of range." should be printed, where X must be replaced by the out-of-range subscript.**
- **The main() function will be given. You should not change the main() function.**

Hints

- To make the program compiled successfully, you must put the array.h and array.cpp together into the same file called array.h and include array.h in main.cpp.
- To make the following two functions, operator<< and operator>>, work correctly,

```
friend ostream &operator<<( ostream &, const Array& );
friend istream &operator>>( istream &, Array & );
```

you must make these two friend functions into function templates with respect to any array class-template specialization. That is, rewriting these two functions into that shown below:

```
template <typename U> friend ostream &operator<<( ostream &, const Array<U> & );
template <typename U> friend istream &operator>>( istream &, Array<U> & );
```

Here, the typename U in template<typename U> should be different from the typename T in template<typename T> for the class Array. This is a must.

Key Points for Grading

- **Output must be correct. Especially, the parts marked by red outlines.**

Main() Function (1)

```
int main()
{
   Array<int> integers1( 5); // seven-element Array
   Array<int> integers2; // 10-element Array by default

   cout << "Enter 15 integers:" << endl;
   cin >> integers1 >> integers2;

   cout << "\nAfter input, the Arrays contain:\n"
      << "integers1: \n" << integers1
      << "integers2: \n" << integers2;

   // use overloaded inequality (!=) operator
   cout << "\nEvaluating: integers1 != integers2" << endl;

   if ( integers1 != integers2 )
      cout << "integers1 and integers2 are not equal" << endl;

   // create Array integers3 using integers1 as an
   // initializer; print size and contents
   Array<double> double1(10); // invokes copy constructor
   cout << "\nEnter 10 double precision numbers:" << endl;
   cin >> double1;

   // use overloaded assignment (=) operator
   cout << "\nCreate double2 initialized with double1: " << endl;
   Array<double> double2(double1); // note target Array is smaller

   cout << "double1: \n" << double1
      << "double2: \n" << double2;
```

Main() Function (2)

```
// use overloaded equality (==) operator
cout << "\nEvaluating: double1 == double2" << endl;

if ( double1 == double2 )
   cout << "double1 and double2 are equal" << endl;
// use overloaded subscript operator to create lvalue
 cout << "\n\nAssigning 100.01 to double1[5]" << endl;
 double1[ 5 ] = 100.01;
 cout << "double1:\n" << double1 << endl;
  double1.swap(11, 0);
   double1.swap(-1, 9);
   cout << "double1[2]: " << double1[2] << "   double1[8]: " << double1[8] <<endl;
   double1.swap(2, 8);
   cout << "After swapping: double1[2]: " << double1[2] << "   double1[8]: " << double1[8] <<endl;

 Array<string> strA(4);
 cout <<"\nEnter 4 strings:" << endl;
 cin >> strA;
 cout << "strA[1]: " << strA[1] << "   strA[3]: " << strA[3] <<endl;
 strA.swap(1, 3);
 cout << "After swapping: strA[1]: " << strA[1] << "   strA[3]: " << strA[3] <<endl;

// attempt to use out-of-range subscript
cout << "\nAttempt to assign \"abcd\" to strA[5]" << endl;
strA[ 5 ] = "abcd"; // ERROR: out of range
return 0;
} // end main
```

Example Input & Output

```
Enter 15 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

After input, the Arrays contain:
integers1:
           1           2           3           4
           5
integers2:
           6           7           8           9
          10          11          12          13
          14          15

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Enter 10 double precision numbers:
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

Create double2 initialized with double1:
double1:
         0.1         0.2         0.3         0.4
         0.5         0.6         0.7         0.8
         0.9           1
double2:
         0.1         0.2         0.3         0.4
         0.5         0.6         0.7         0.8
         0.9           1

Evaluating: double1 == double2
double1 and double2 are equal
```

Example Input & Output (cont.)

```
Assigning 100.01 to double1[5]
double1:
         0.1          0.2          0.3          0.4
         0.5       100.01          0.7          0.8
         0.9            1

** Error: swapping fails,  subscript 11 out of range.
** Error: swapping fails,  subscript -1 out of range.
double1[2]: 0.3   double1[8]: 0.9
After swapping: double1[2]: 0.9   double1[8]: 0.3

Enter 4 strings:
Taiwan has great people.
strA[1]: has   strA[3]: people.
After swapping: strA[1]: people.   strA[3]: has

Attempt to assign "abcd" to strA[5]

Error: Subscript 5 is out of range
```