Chapter 14 Templates

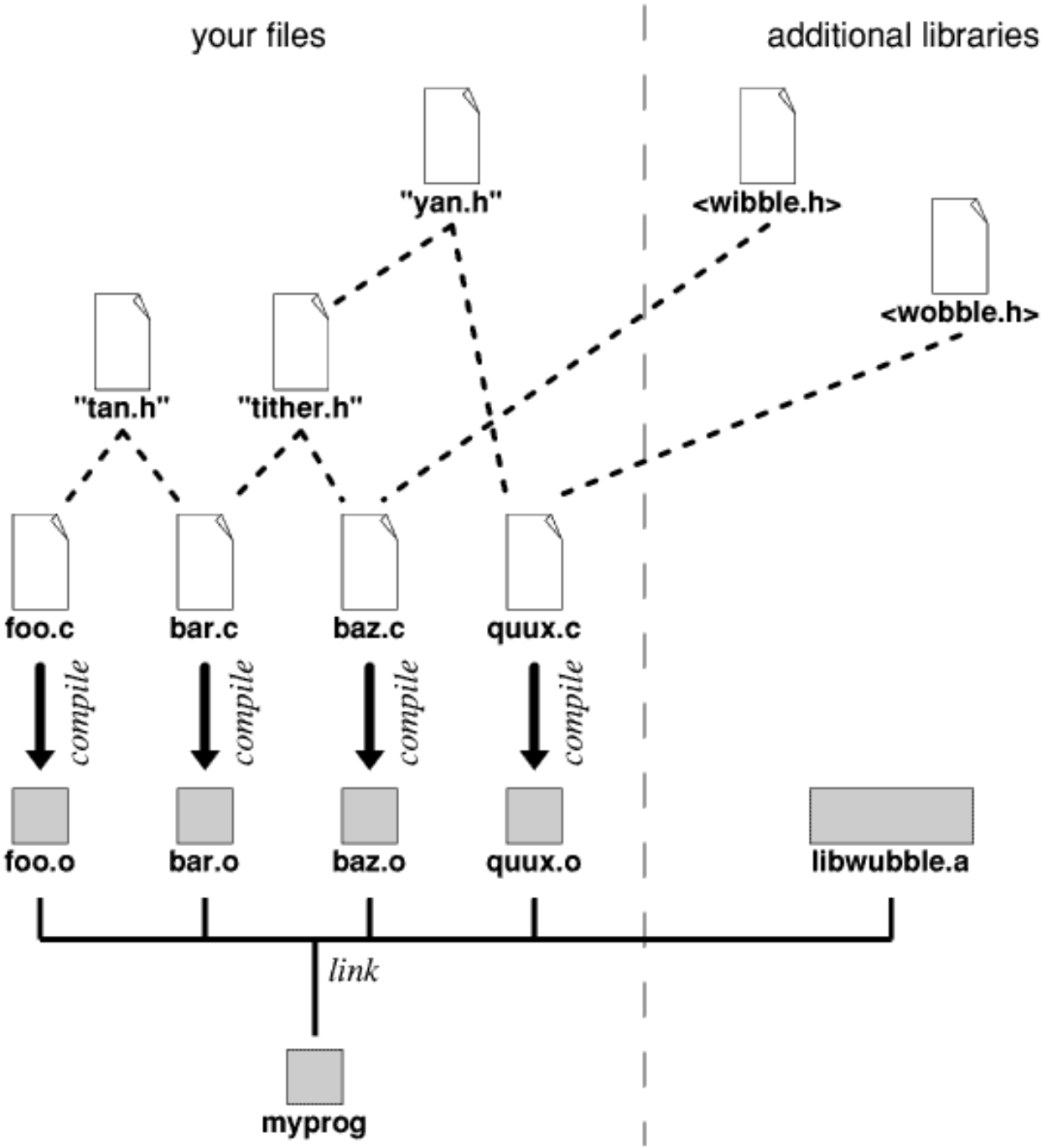
Yu-Shuen Wang, CS, NCTU

14.1 Introduction

- Function templates and class templates enable you to specify, with a single code segment, an entire range of related functions.
- This technique is called generic programming.

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 `#include`*d into the appropriate client source-code files. For class templates, this means that the member functions are also defined in the header file.*


your files
additional libraries
"yan.h"
<wibble.h>
<wobble.h>
"tan.h"
"tither.h"
foo.c
bar.c
baz.c
quux.c
compile
compile
compile
compile
foo.o
bar.o
baz.o
quux.o
libwubble.a
link
myprog

14.2 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 to handle each function call appropriately.

14.2 Function Templates (cont.)

- All function-template definitions look like
  - `template<typename T>`
  - `template<class ElementType>`
  - `template<typename BorderType, typename FillType>`
- The template parameters are used to specify the types of the
  - arguments to the function
  - return type of the function
  - variables within the function.
- Keywords `typename` and `class` mean "any fundamental type or user-defined type."

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.*

```
1  // Fig. 14.1: fig14_01.cpp
2  // Using template functions.
3  #include <iostream>
4  using namespace std;
5
6  // function template printArray definition
7  template< typename T >
8  void printArray( const T * const array, int count )
9  {
10    for ( int i = 0; i < count; i++ )
11       cout << array[ i ] << " ";
12
13    cout << endl;
14 } // end function template printArray
15
16 int main()
17 {
18    const int aCount = 5; // size of array a
19    const int bCount = 7; // size of array b
20    const int cCount = 6; // size of array c
21
```

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

```
22      int a[ aCount ] = { 1, 2, 3, 4, 5 };
23      double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
24      char c[ cCount ] = "HELLO"; // 6th position for null
25
26      cout << "Array a contains:" << endl;
27
28      // call integer function-template specialization
29      printArray( a, aCount );
30
31      cout << "Array b contains:" << endl;
32
33      // call double function-template specialization
34      printArray( b, bCount );
35
36      cout << "Array c contains:" << endl;
37
38      // call character function-template specialization
39      printArray( c, cCount );
40  } // end main
```

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
```

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

```
int a[5] = {1, 2, 3, 4, 5};

// p points to a constant array
const int *p = a;

// p is a constant
int * const p = a;

// p is a constant and it points to a constant array
const int * const p = a;
```

```
int a[5] = {1, 2, 3, 4, 5};

// p points to a constant array
const int *p = a; (p++; acceptable)

// p is a constant
int * const p = a; (*p=2; acceptable)

// p is a constant and it points to a constant array
const int * const p = a; (p++; *p=2; non-acceptable)
```

14.2 Function Templates (cont.)

- The function-template specialization for type `int` is
  - ```
    void printArray( const int * const array, int count )
    {
       for ( int i = 0; i < count; i++ )
          cout << array[ i ] << " ";

       cout << endl;
    } // end function printArray
    ```
- As with function parameters, the names of template parameters must be unique inside a template definition.
- Template parameter names need not be unique across different function templates.
    ```

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.

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

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

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 templates are 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 you'd have written to produce the separate overloaded functions.

14.3 Overloading Function Templates

- Function templates and overloading are intimately related.
- The function-template specializations generated from a function template all have the same name, so the compiler uses overloading resolution to invoke the proper function.

14.4 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
- To instantiate a stack, a data type must be specified.
- Wonderful opportunity for software reusability.
- C++ provides this capability through class templates.

Software Engineering Observation 14.2

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

14.4 Class Templates (cont.)

- 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.

- Each time an additional class-template specialization is needed, you use a concise, simple notation, and the compiler writes the source code for the specialization you require.

14.4 Class Templates (cont.)

- We simply write
  - `template< typename T >`
- to specify a class-template definition with type parameter `T`.
- Due to the way this class template is designed, there are two constraints for nonfundamental data types used with this `Stack` `(dynamic memory)`
  - they must have a default constructor
  - their assignment operators must properly copy objects into the `Stack`

```
1  // Fig. 14.2: Stack.h
2  // Stack class template.
3  #ifndef STACK_H
4  #define STACK_H
5
6  template< typename T >
7  class Stack
8  {
9  public:
10    Stack( int = 10 ); // default constructor (Stack size 10)
11
12    // destructor
13    ~Stack()
14    {
15       delete [] stackPtr; // deallocate internal space for Stack
16    } // end ~Stack destructor
17
18    bool push( const T & ); // push an element onto the Stack
19    bool pop( T & ); // pop an element off the Stack
20
```

Fig. 14.2 | Class template `Stack`. (Part 1 of 4.)

```
21      // determine whether Stack is empty
22      bool isEmpty() const
23      {
24         return top == -1;
25      } // end function isEmpty
26
27      // determine whether Stack is full
28      bool isFull() const
29      {
30         return top == size - 1;
31      } // end function isFull
32
33   private:
34      int size; // # of elements in the Stack
35      int top; // location of the top element (-1 means empty)
36      T *stackPtr; // pointer to internal representation of the Stack
37   }; // end class template Stack
38
```

Fig. 14.2 | Class template `Stack`. (Part 2 of 4.)

```
39  // constructor template
40  template< typename T >
41  Stack< T >::Stack( int s )
42     : size( s > 0 ? s : 10 ), // validate size
43       top( -1 ), // Stack initially empty
44       stackPtr( new T[ size ] ) // allocate memory for elements
45  {
46     // empty body
47  } // end Stack constructor template
48
49  // push element onto Stack;
50  // if successful, return true; otherwise, return false
51  template< typename T >
52  bool Stack< T >::push( const T &pushValue )
53  {
54     if ( !isFull() )
55     {
56        stackPtr[ ++top ] = pushValue; // place item on Stack
57        return true; // push successful
58     } // end if
59
60     return false; // push unsuccessful
61  } // end function template push
62
```

Fig. 14.2 | Class template `Stack`. (Part 3 of 4.)

```
63   // pop element off Stack;
64   // if successful, return true; otherwise, return false
65   template< typename T >
66   bool Stack< T >::pop( T &popValue )
67   {
68      if ( !isEmpty() )
69      {
70         popValue = stackPtr[ top-- ]; // remove item from Stack
71         return true; // pop successful
72      } // end if
73
74      return false; // pop unsuccessful
75   } // end function template pop
76
77   #endif
```

Fig. 14.2 | Class template `Stack`. (Part 4 of 4.)

14.4 Class Templates (cont.)

- Although templates offer software-reusability benefits, remember that multiple class-template specializations are instantiated in a program (at compile time), even though the template is written only once.

```
1  // Fig. 14.3: fig14_03.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( 5 ); // size 5
10    double doubleValue = 1.1;
11
12    cout << "Pushing elements onto doubleStack\n";
13
14    // push 5 doubles onto doubleStack
15    while ( doubleStack.push( doubleValue ) )
16    {
17       cout << doubleValue << ' ';
18       doubleValue += 1.1;
19    } // end while
20
21    cout << "\nStack is full. Cannot push " << doubleValue
22       << "\n\nPopping elements from doubleStack\n";
23
```

Fig. 14.3 | Class template Stack `test` program. (Part 1 of 3.)

```
24    // pop elements from doubleStack
25    while ( doubleStack.pop( doubleValue ) )
26       cout << doubleValue << ' ';
27
28    cout << "\nStack is empty. Cannot pop\n";
29
30    Stack< int > intStack; // default size 10
31    int intValue = 1;
32    cout << "\nPushing elements onto intStack\n";
33
34    // push 10 integers onto intStack
35    while ( intStack.push( intValue ) )
36    {
37       cout << intValue++ << ' ';
38    } // end while
39
40    cout << "\nStack is full. Cannot push " << intValue
41       << "\n\nPopping elements from intStack\n";
42
43    // pop elements from intStack
44    while ( intStack.pop( intValue ) )
45       cout << intValue << ' ';
46
47    cout << "\nStack is empty. Cannot pop" << endl;
48 } // end main
```

Fig. 14.3 | Class template Stack `test` program. (Part 2 of 3.)

```
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
```

Fig. 14.3 | Class template Stack `test` program. (Part 3 of 3.)

```
1  // Fig. 14.4: fig14_04.cpp
2  // Stack class template test program. Function main uses a
3  // function template to manipulate objects of type Stack< T >.
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    T value, // initial value to push
14    T increment, // increment for subsequent values
15    const string stackName ) // name of the Stack< T > object
16 {
17    cout << "\nPushing elements onto " << stackName << '\n';
18
19    // push element onto Stack
20    while ( theStack.push( value ) )
21    {
22       cout << value << ' ';
23       value += increment;
24    } // end while
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 1 of 3.)

```
25
26       cout << "\nStack is full. Cannot push " << value
27          << "\n\nPopping elements from " << stackName << '\n';
28
29       // pop elements from Stack
30       while ( theStack.pop( value ) )
31          cout << value << ' ';
32
33       cout << "\nStack is empty. Cannot pop" << endl;
34    } // end function template testStack
35
36    int main()
37    {
38       Stack< double > doubleStack( 5 ); // size 5
39       Stack< int > intStack; // default size 10
40
41       testStack( doubleStack, 1.1, 1.1, "doubleStack" );
42       testStack( intStack, 1, 1, "intStack" );
43    } // end main
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 2 of 3.)

```
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
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 3 of 3.)

14.5 Nontype Parameters and Default Types for Class Templates

- It's also possible to use non-type template parameters. 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 a 100-element `Stack` class-template specialization of `double` values named `mostRecentSalesFigures`.

```
1  // Fig. 14.2: Stack.h
2  // Stack class template.
3  #ifndef STACK_H
4  #define STACK_H
5
6  template< typename T >
7  class Stack
8  {
9  public:
10    Stack( int = 10 ); // default constructor (Stack size 10)
11
12    // destructor
13    ~Stack()
14    {
15       delete [] stackPtr; // deallocate internal space for Stack
16    } // end ~Stack destructor
17
18    bool push( const T & ); // push an element onto the Stack
19    bool pop( T & ); // pop an element off the Stack
20
```

```
template< typename T, int elements >
class Stack
{
public:
        Stack( int = elements);

...
}
```

Fig. 14.2 | Class template Stack. (Part 1 of 4.)

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

- 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 `jobDescriptions`; this class-template specialization would be of type `Stack< string >`.

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.*

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

- If a particular user-defined type will not work with a template or requires customized processing, you can define an explicit specialization of the class template for a particular type.

- For example, form a new class with the name `Stack< Employee >` as follows:
  - ```
    template<>
    class Stack< Employee >
    {   // body of class definition
    };
    ```
    ```

14.6 Notes on Templates and Inheritance

- Templates and inheritance relate in several ways:
  - 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.

```
class Base
{
        // Generic, non-type-specific code
};

template <typename T>
class TypedRealBase : public Base
{
      // A class template derived from a nontemplate class
};

template <typename T>
class TypedBase : public TypedRealBase<Employ>
{
      // A class template derived from a class-template specialization
};
```

```
template <>
class TypedBase<Employ> : public TypedRealBase<Employ>
{
        // A class-template specialization derived from a class-template
            specialization
};

// template <>
class TypedBase : public TypedRealBase<Employ>
{
        // A nontemplate class derived from a class-template specialization
};
```

14.7 Notes on Templates and Friends

- With class templates, friendship can be established between a class template and a global function, a member function of another class, or even an entire class.

member function of another class

```
class B;

class A {
public:
        int Func1( B& b );
private:
        int Func2( B& b );
};

class B {
private:
        int _b;

        // A::Func1 is a friend function to class B
        // so A::Func1 has access to all members of B

        friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }   // OK
int A::Func2( B& b ) { return b._b; }   // Compile error!
```

friend class

```
class Texture
{
public:
        friend class JpegLoader;

        uint width() const { return mWidth; }
        uint height() const { return mHeight; }
private:
        uint mWidth;
        uint mHeight;
};

class JpegLoader
{
        Texture mTexture;

        void load()
        {
                // ...
                mTexture.mWidth = 128;
        }
};
```

14.8 Notes on Templates and `static` Members

- Each class-template specialization instantiated from a class template has its own copy of each `static` data member of the class template; all objects of that specialization share that one `static` data member.
- In addition, as with `static` data members of nontemplate classes, `static` data members of class-template specializations must be defined and, if necessary, initialized at global namespace scope.
- Each class-template specialization gets its own copy of the class template's `static` member functions.