Cpt S 122 – Data Structures

Templates

Nirmalya Roy

School of Electrical Engineering and Computer Science
Washington State University

Topics

- Introduction
- Function Template
  - Function-template and function-template specializations
- Overloading Function Template
- Class Templates
  - Class-template and class-template specializations
  - STL containers: example of container class template such as stack
- Non-type Parameters & Default Types of Class Templates

Introduction

- Function templates and class templates enable to specify, with a single code segment,
  - an entire range of related (overloaded) functions
    - function-template specializations
  - an entire range of related classes
    - class-template specializations.
- This technique is called generic programming.
- Note the distinction 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.*

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.)

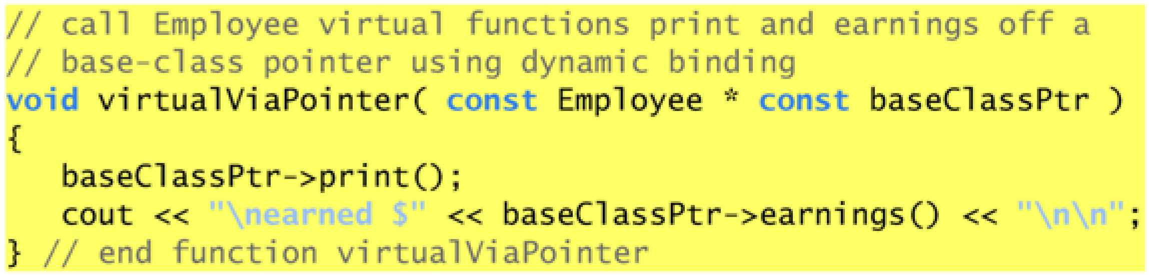
- All function-template definitions begin with keyword `template` followed by a list of template parameters to the function template enclosed in angle brackets (< and >);
- Each template parameter that represents a type must be preceded by either of the interchangeable keywords `class` or `typename`, as in
  - `template<typename T>`
  - `template<class ElementType>`
  - `template<typename BorderType, typename FillType>`
- The type template parameters 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
  - "*any fundamental type or user-defined type*."

```cpp
// Fig. 14.1: fig14_01.cpp
// Using function-template specializations.
#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

```

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

Function Templates (cont.)

- Function template `printArray`
  - declares a single template parameter `T` (`T` can be any valid identifier) for the type of the array;
  - `T` is referred to as a type template parameter, or type parameter.



```
// call Employee virtual functions print and earnings off a
// base-class pointer using dynamic binding
void virtualViaPointer( const Employee * const baseClassPtr )
{
   baseClassPtr->print();
   cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
} // end function virtualViaPointer
```

```
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;

```

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

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

```
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.)

Function Templates (cont.)

- When the compiler detects a `printArray` function invocation in the client program,
  - the compiler uses its *overload resolution capabilities* to find a definition of function `printArray` that best matches the function call.
- The compiler compares the type of `printArray`'s first argument (`int *`) to the `printArray` function template's first parameter (`const T * const`)
  - deduces that replacing the type parameter `T` with `int` would make the argument consistent with the parameter.
- The compiler substitutes `int` for `T` throughout the template definition
  - compiles a `printArray` specialization that can display an array of `int` values.

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.
- It’s important to note that if `T` represents a user-defined type,
  - there must be an *overloaded stream insertion operator* for that type; otherwise,
  - the first stream insertion operator will not compile.

Observations

- If a template is invoked with a user-defined types and if that template uses functions or operators (= ; <= ; == etc) with objects of that class type
  - those functions or operators must be overloaded for the user-defined type,
  - otherwise such operator causes *compilation errors*
- Templates offer software reusability benefits
  - Templates are written once
  - Multiple function-template specializations and class-template specializations are instantiated in a program (at compile time)
  - Code generated by template is of the same size of code in case of separate overloaded functions

Overloading Function Templates

- *Function templates* and *overloading* are intimately related.
- The function-template specializations generated from a function template all have the same name,
  - compiler uses overloading resolution to invoke the proper function.
- A function template may be overloaded in several ways.
  - We can provide other function templates that specify the same function name but *different function parameters*.
  - We can provide *nontemplate functions* with the same function name but different function arguments.

Observations: More than one type parameter

- Function templates may accept more than one type parameter,
  - simply by specifying more template parameters between the angle brackets. For example:
- `template <class T, class U>`

```
T GetMin (T a, U b) {
return (a < b ? a : b);
}
```

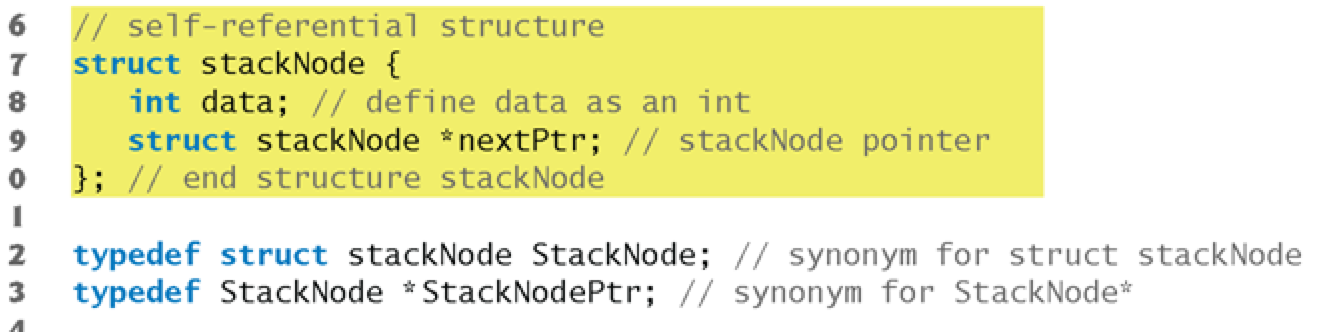
  - function template GetMin() accepts two parameters of different types
  - returns an object of the same type as the first parameter (T) that is passed

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.
- Wonderful opportunity for software reusability.
- Need the means for describing the notion of a stack generically
  - instantiating classes that are type-specific versions of this generic stack class.
- C++ provides this capability through *class templates*.
- Class templates encourage s/w reusability by enabling type-specific versions of generic classes to be instantiated.

Class Templates (cont.)

- Class templates are called parameterized types
  - they require one or more type parameters to specify how to customize a “generic class” template to form a class-template specialization.
  - When an additional specialization is needed, you use a concise, simple notation, and the compiler writes the source code for that specialization.



```
// self-referential structure
struct stackNode {
   int data; // define data as an int
   struct stackNode *nextPtr; // stackNode pointer
}; // end structure stackNode

typedef struct stackNode StackNode; // synonym for struct stackNode
typedef StackNode *StackNodePtr; // synonym for StackNode*
```

Class Templates (cont.)

- `Stack` class-template
- It looks like a conventional class definition, except that it's preceded by the header
    - `template< typename T >`
    - specify a class-template definition with type parameter `T` which acts as a placeholder for the type of the `Stack` class to be created.
- The *type of element to be stored on this `Stack` is mentioned generically as `T`*
    - used throughout the `Stack` class header and member-function definitions.
- Due to the way this class template is designed, there are two constraints for nonfundamental data types used with this `Stack`
    - they must have a default constructor
    - their assignment operators must properly copy objects into the `Stack`

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

template< typename T >
class Stack
{
public:
   explicit 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
```

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

```

   // 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

```

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

```cpp
// 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

```

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

```
// 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. 14.2 | Stack class template. (Part 4 of 4.)

Class Templates (cont.)

- The member-function definitions of a class template are function templates.
- The member-function definitions that appear outside the class template definition each begin with the header
    - `template< typename T >`
  - each definition resembles a conventional function definition,
  - the `Stack` element type always is listed generically as type parameter `T`.
- The binary scope resolution operator is used with the class-template name to tie each member-function definition to the class template's scope.
- When `doubleStack` is instantiated as type `Stack<double>`, the `Stack` constructor function-template specialization uses `new` to create an array of elements of type `double` to represent the stack.

Class Templates (cont.)

- Now, let's consider the driver that exercises the `Stack` class template.
- The driver begins by instantiating object `doubleStack` of size `5`.
- This object is declared to be of class `Stack< double >` (pronounced "`Stack` of `double`").
- The compiler associates type `double` with type parameter `T` in the class template to produce the source code for a `Stack` class of type `double`.
- 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.

```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";

```

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

```cpp
   // 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";

```

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

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

   cout << "\nStack is empty. Cannot pop" << endl;
} // 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
```

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

Class Templates (cont.)

- Instantiates integer stack `intStack` with the declaration
  - `Stack< int > intStack;`
- Because no size is specified, the size defaults to `10` as specified in the default constructor.

Class Templates (cont.)

- `double-Stack` manipulations and the `intStack` manipulations code are *almost identical*
  - present another opportunity to use a function template.
- Define function template `testStack` to perform the same tasks
  - `push` a series of values onto a `Stack< T >`
  - `pop` the values off a `Stack< T >`.
- Function template `testStack` uses template parameter `T` to represent the data type stored in the `Stack< T >`.
- The function template takes four arguments
  - a reference to an object of type `Stack< T >`,
  - a value of type `T` that will be the first value `push`ed onto the `Stack< T >`,
  - a value of type `T` used to increment the values `push`ed onto the `Stack< T >`
  - a `string` that represents the name of the `Stack< T >` object for output purposes.

```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>
#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 >
   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';

```

Fig. 14.4 | Passing a Stack template object to a function template. (Part 1 of 4.)

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

   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

```

Fig. 14.4 | Passing a Stack template object to a function template. (Part 2 of 4.)

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

Fig. 14.4 | 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
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 4 of 4.)

Nontype Parameters and Default Types for Class Templates

- Class template `Stack` used only a type parameter in the template header.
- 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 >`
  - `Stack<double,100> mostRecentSalesFigures;`
  - 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 >`.

Observations

- Specify the size of a container class (such as stack class, or an array) at compile time (possibly through a nontype template parameter); if appropriate.
  - Eliminates the execution time overhead of using `new` to create the space dynamically.
  - Specifying the size of a container at compile time avoids execution time error if `new` is unable to obtain the needed memory.

Nontype 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.
    - `Stack<> jobDescriptions;`
  - instantiate a `Stack` class-template specialization of `string`s named `jobDescriptions`;
  - 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.

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,
  - define an explicit specialization of the class template for a particular type.
- Assume we want to create an explicit specialization `Stack` for `Employee` objects.
- To do this, form a new class with the name `Stack< Employee >` as follows:

  -
    ```
    template<>
    class Stack< Employee >
    {
       // body of class definition
    };
    ```
- The `Stack<Employee>` explicit specialization is a complete replacement for the `Stack` class template that is specific to type `Employee`.

C++11 Standard: Variadic Templates

- **Variadic template** is a template, which can take an arbitrary number of template arguments of any type.
  - Both the classes & functions can be variadic
- Variadic class template:
  - *template*<*typename*... Arguments>
  - *class* VariadicTemplate;
- An instance of this class template
  - VariadicTemplate<*double, float*> instance;
  - VariadicTemplate<*bool, unsigned short int, long*> instance;
  - VariadicTemplate<*char*, std::vector<*int*>, std::string, std::vector<*long long*>> instance;