Lesson 7 - Templates

Outline

1 Introduction

2 Templates Functions

3 Overloading Template Functions

4 Class Templates

5 Class Templates and Non-type Parameters

6 Templates and friends

7 Templates and static Members

◀ ▶

Introduction

- Templates:
  - **easily create a large range of related functions or classes**
  - the blueprint of the related functions
  - perform identical operations on different data types
  - provide type checking
- Template statement:
  - can use **`class`** or **`typename`** - specifies type parameters
  - **`template`** statement:

```
template<class type, class type...>
```

  - Function definition follows template statements

An example of template function

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

   cout << endl;
}
```

T is the type parameter. **T**'s type is detected and substituted inside the function.

The newly created function is compiled.

The `int` version of `printArray` is

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

   cout << endl;
}
```

```cpp
// Fig 12.2: fig12_02.cpp
// Using template functions
#include <iostream>

using std::cout;
using std::endl;

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

   cout << endl;
}

int main()
{
   const int aCount = 5, bCount = 7, cCount = 6;
   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;
   printArray( a, aCount );  // integer template function

   cout << "Array b contains:" << endl;
   printArray( b, bCount );  // double template function

   cout << "Array c contains:" << endl;
   printArray( c, cCount );  // character template function

   return 0;
}
```

Notice how type parameter **`T`** is used in place of **`int`**, **`float`**, etc. .

Each type array gets operated on by a different template function.

Outline

1. Function template definition

1.1 Initialize variables

2. Call template functions

3. Output

```
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
```

Program Output

Overloading Template Functions

- template functions can be overloaded
- related template functions have same name
  - compiler uses overloading resolution to call the right one
- compiler tries to match function call with function name and arguments
  - if no precise match, looks for function templates
    - if found, compiler generates and uses template function
  - if no matches or multiple matches are found, compiler gives error

Class Templates

- class templates
  - allow type-specific versions of generic classes
- Format:

```
template <class T>
class ClassName{
 definition
 }
```

  - Need not use **"T"**, any identifier will work
  - To create an object of the class, type

    `ClassName< type > myObject;`

    Example: **`Stack< double > doubleStack;`**

Class Templates (II)

- Template class functions
  - declared normally, but preceded by **`template<class T>`**
    - generic data in class listed as type **`T`**
  - Template class function definition:
    - constructor definition - creates an array of type **`T`**

```
template<class T>
MyClass< T >::MyClass(int size)
{
   myArray = new T[size];
}
```

```
// Fig. 12.3: tstack1.h
// Class template Stack
#ifndef TSTACK1_H
#define TSTACK1_H

template< class T >
class Stack {
public:
   Stack( int = 10 );    // default constructor (stack size 10)
   ~Stack() { delete [] stackPtr; } // destructor
   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
   T *stackPtr;          // pointer to the stack

   bool isEmpty() const { return top == -1; }      // utility
   bool isFull() const { return top == size - 1; } // functions
};

// Constructor with default size 10
template< class T >
Stack< T >::Stack( int s )
{
   size = s > 0 ? s : 10;
   top = -1;               // Stack is initially empty
   stackPtr = new T[ size ]; // allocate space for elements
}
```

Notice how a member function of the class template is defined

1. Class template definition

1.1 Function definitions

1.2 Stack constructor

```

// Push an element onto the stack
// return 1 if successful, 0 otherwise
template< class T >
bool Stack< T >::push( const T &pushValue )
{
   if ( !isFull() ) {
      stackPtr[ ++top ] = pushValue; // place item in Stack
      return true;  // push successful
   }
   return false;     // push unsuccessful
}

// Pop an element off the stack
template< class T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() ) {
      popValue = stackPtr[ top-- ];  // remove item from Stack
      return true;  // pop successful
   }
   return false;     // pop unsuccessful
}

#endif
```

Test if the stack is full. If not, push element.

Test if the stack is empty. If not, pop an element.

1.3 push

1.4 pop

```
// Fig. 12.3: fig12_03.cpp
// Test driver for Stack template
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "tstack1.h"

int main()
{
   Stack< double > doubleStack( 5 );
   double f = 1.1;
   cout << "Pushing elements onto doubleStack\n";

   while ( doubleStack.push( f ) ) { // success true returned
      cout << f << ' ';
      f += 1.1;
   }

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

   while ( doubleStack.pop( f ) )  // success true returned
```

```
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
```

1. Load header

1.1 Initialize doubleStack

1.2 Initialize variables

2. Function calls

```
      cout << f << ' ';

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

   Stack< int > intStack;
   int i = 1;
   cout << "\nPushing elements onto intStack\n";

   while ( intStack.push( i ) ) { // success true returned
      cout << i << ' ';
      ++i;
   }

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

   while ( intStack.pop( i ) )  // success true returned
      cout << i << ' ';

   cout << "\nStack is empty. Cannot pop\n";
   return 0;
}
```

```
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
```

Outline

2. Function calls

3. Output

Program Output

```
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
```

Class Templates and Non-type Parameters

- Non-type parameters in templates
  - default argument
  - treated as **`const`**
- Example:

```
template< class T, int elements >
Stack< double, 100 > mostRecentSalesFigures;
```

  - declares object of type **`Stack< double, 100>`**
  - This may appear in the class definition:

```
T stackHolder[ elements ]; //array to hold stack
```

  - creates array at compile time, rather than dynamic allocation at execution time
- Template classes can be overridden
  - The template remains for unoverriden types

Templates and friends

- friendships allowed between a class template and
  - global function
  - member function of another class
  - entire class
- **`friend`** functions
  - inside definition of class template **`X`**:
  - **`friend void f1();`**
    - **`f1()`** a **`friend`** of *all* template classes
  - **`friend void f2( X< T > & );`**
    - **`f2( X< int > & )`** is a **`friend`** of **`X< int >`** *only*.
    - The same applies for **`float`**, **`double`**, etc.
  - **`friend void A::f3();`**
    - member function **`f3`** of class **`A`** is a **`friend`** of all template classes

Templates and friends (II)

  - **`friend void C< T >::f4( X< T > & );`**
    - **`C<float>::f4( X< float> & )`** is a **`friend`** of **`class X<float>`** only

- **`friend`** classes
  - **`friend class Y;`**
    - every member function of **`Y`** a friend with every template class made from **`X`**
  - **`friend class Z<T>;`**
    - class **`Z<float>`** a **`friend`** of class **`X<float>,`** etc.

Templates and static Members

- non-template class
  - **`static`** data members shared between all objects

- template classes
  - each class (**`int`**, **`float`**, etc.) has its own copy of **`static`** data members
  - **`static`** variables initialized at file scope
  - each template class gets its own copy of **`static`** member functions