Templates

D&D Chapter 12

Introduction

- Templates - easily create a large range of related functions or classes
  - function template - the blueprint of the related functions
  - template function - a specific function *made* from a function template

Function Templates

- overloaded functions
  - perform similar operations on different data types
- function templates
  - perform identical operations on different data types
  - provide type checking
- Format:

  `template<class` *`type`*`, `**`class`** *`type`*`...>`
  - can use **`class`** or **`typename`** - specifies type parameters

  ```
  template< class T >
  template< typename ElementType >
  template< class BorderType, class FillType >
  ```
  - Function definition follows **`template`** statement

```
1  template< class T >

2  void printArray( const T *array, const int
count )
3  {

4     for ( int i = 0; i < count; i++ )

5        cout << array[ i ] << " ";

6

7     cout << endl;

8  }
```

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;
}
```

```
 1 // Fig 12.2: fig12_02.cpp
 2 // Using template functions
 3 #include <iostream>
 4
 5 using std::cout;
 6 using std::endl;
 7
 8 template< class T >
 9 void printArray( const T *array, const int count )
10 {
11    for ( int i = 0; i < count; i++ )
12       cout << array[ i ] << " ";
13
14    cout << endl;
15 }
16
17 int main()
18 {
19    const int aCount = 5, bCount = 7, cCount = 6;
20    int a[ aCount ] = { 1, 2, 3, 4, 5 };
21    double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
22    char c[ cCount ] = "HELLO";  // 6th position for null
23
24    cout << "Array a contains:" << endl;
25    printArray( a, aCount );  // integer template function
26
27    cout << "Array b contains:" << endl;
28    printArray( b, bCount );  // double template function
29
30    cout << "Array c contains:" << endl;
31    printArray( c, cCount );  // character template function
32
33    return 0;
34 }
```

```
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

- related template functions have same name
  - compiler uses overloading resolution to call the right one
- function template can be overloaded
  - other function templates can have same name but different number of parameters
  - non-template function can have same name but different arguments
- 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: List`< double > doubleList;`

Class Templates (II)

- Template class functions
  - declared normally, but preceded by **`template<class T>`**
    - generic data in class listed as type **`T`**
  - binary scope resolution operator used
  - Template class function definition:

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

    - constructor definition - creates an array of type **`T`**

```
1  // Fig. 15.3: listnd.h
2  // ListNode template definition
3  #ifndef LISTND_H
4  #define LISTND_H
5
6  template< class NODETYPE > class List;  // forward declaration
7
8  template<class NODETYPE>
9  class ListNode {
10    friend class List< NODETYPE >; // make List a friend
11 public:
12    ListNode( const NODETYPE & );  // constructor
13    NODETYPE getData() const;      // return data in the node
14 private:
15    NODETYPE data;                 // data
16    ListNode< NODETYPE > *nextPtr; // next node in the list
17 };
18
19 // Constructor
20 template<class NODETYPE>
21 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
22    : data( info ), nextPtr( 0 ) { }
23
24 // Return a copy of the data in the node
25 template< class NODETYPE >
26 NODETYPE ListNode< NODETYPE >::getData() const { return data; }
27
28 #endif
```

```
// Fig. 15.3: list.h
// Template List class definition
#ifndef LIST_H
#define LIST_H

#include <iostream>
#include <cassert>
#include "listnd.h"

using std::cout;

template< class NODETYPE >
class List {
public:
   List();      // constructor
   ~List();     // destructor
   void insertAtFront( const NODETYPE & );
   void insertAtBack( const NODETYPE & );
   bool removeFromFront( NODETYPE & );
   bool removeFromBack( NODETYPE & );
   bool isEmpty() const;
   void print() const;
private:
   ListNode< NODETYPE > *firstPtr;  // pointer to first node
   ListNode< NODETYPE > *lastPtr;   // pointer to last node

   // Utility function to allocate a new node
   ListNode< NODETYPE > *getNewNode( const NODETYPE & );
};

// Default constructor
template< class NODETYPE >
List< NODETYPE >::List() : firstPtr( 0 ), lastPtr( 0 ) { }
```

```

// Destructor
template< class NODETYPE >
List< NODETYPE >::~List()
{
   if ( !isEmpty() ) {    // List is not empty
      cout << "Destroying nodes ...\n";

      ListNode< NODETYPE > *currentPtr = firstPtr, *tempPtr;

      while ( currentPtr != 0 ) {  // delete remaining nodes
         tempPtr = currentPtr;
         cout << tempPtr->data << '\n';
         currentPtr = currentPtr->nextPtr;
         delete tempPtr;
      }
   }

   cout << "All nodes destroyed\n\n";
}

// Insert a node at the front of the list
template< class NODETYPE >
void List< NODETYPE >::insertAtFront( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value );

   if ( isEmpty() )  // List is empty
      firstPtr = lastPtr = newPtr;
   else {          // List is not empty
      newPtr->nextPtr = firstPtr;
      firstPtr = newPtr;
   }
}
```

```
96
97 // Insert a node at the back of the list
98 template< class NODETYPE >
99 void List< NODETYPE >::insertAtBack( const NODETYPE &value )
100{
101   ListNode< NODETYPE > *newPtr = getNewNode( value );
102
103   if ( isEmpty() )  // List is empty
104      firstPtr = lastPtr = newPtr;
105   else {          // List is not empty
106      lastPtr->nextPtr = newPtr;
107      lastPtr = newPtr;
108   }
109}
110
111// Delete a node from the front of the list
112template< class NODETYPE >
113bool List< NODETYPE >::removeFromFront( NODETYPE &value )
114{
115   if ( isEmpty() )             // List is empty
116      return false;             // delete unsuccessful
117   else {
118      ListNode< NODETYPE > *tempPtr = firstPtr;
119
120      if ( firstPtr == lastPtr )
121         firstPtr = lastPtr = 0;
122      else
123         firstPtr = firstPtr->nextPtr;
124
125      value = tempPtr->data;  // data being removed
126      delete tempPtr;
127      return true;            // delete successful
128   }
129}
```

```
130
131// Delete a node from the back of the list
132template< class NODETYPE >
133bool List< NODETYPE >::removeFromBack( NODETYPE &value )
134{
135   if ( isEmpty() )
136      return false;   // delete unsuccessful
137   else {
138      ListNode< NODETYPE > *tempPtr = lastPtr;
139
140      if ( firstPtr == lastPtr )
141         firstPtr = lastPtr = 0;
142      else {
143         ListNode< NODETYPE > *currentPtr = firstPtr;
144
145         while ( currentPtr->nextPtr != lastPtr )
146            currentPtr = currentPtr->nextPtr;
147
148         lastPtr = currentPtr;
149         currentPtr->nextPtr = 0;
150      }
151
152      value = tempPtr->data;
153      delete tempPtr;
154      return true;   // delete successful
155   }
156}
157
158// Is the List empty?
159template< class NODETYPE >
160bool List< NODETYPE >::isEmpty() const
161   { return firstPtr == 0; }
162
163// return a pointer to a newly allocated node
```

```
164 template< class NODETYPE >
165 ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
166                                       const NODETYPE &value )
167 {
168    ListNode< NODETYPE > *ptr =
169       new ListNode< NODETYPE >( value );
170    assert( ptr != 0 );
171    return ptr;
172 }
173
174 // Display the contents of the List
175 template< class NODETYPE >
176 void List< NODETYPE >::print() const
177 {
178    if ( isEmpty() ) {
179       cout << "The list is empty\n\n";
180       return;
181    }
182
183    ListNode< NODETYPE > *currentPtr = firstPtr;
184
185    cout << "The list is: ";
186
187    while ( currentPtr != 0 ) {
188       cout << currentPtr->data << ' ';
189       currentPtr = currentPtr->nextPtr;
190    }
191
192    cout << "\n\n";
193 }
194
195 #endif
```

```
196 // Fig. 15.3: fig15_03.cpp
197 // List class test
198 #include <iostream>
199 #include "list.h"
200
201 using std::cin;
202 using std::endl;
203
204 // Function to test an integer List
205 template< class T >
206 void testList( List< T > &listObject, const char *type )
207 {
208    cout << "Testing a List of " << type << " values\n";
209
210    instructions();
211    int choice;
212    T value;
213
214    do {
215       cout << "? ";
216       cin >> choice;
217
218       switch ( choice ) {
219          case 1:
220             cout << "Enter " << type << ": ";
221             cin >> value;
222             listObject.insertAtFront( value );
223             listObject.print();
224             break;
225          case 2:
226             cout << "Enter " << type << ": ";
227             cin >> value;
```

```
228            listObject.insertAtBack( value );
229            listObject.print();
230            break;
231         case 3:
232            if ( listObject.removeFromFront( value ) )
233               cout << value << " removed from list\n";
234
235            listObject.print();
236            break;
237         case 4:
238            if ( listObject.removeFromBack( value ) )
239               cout << value << " removed from list\n";
240
241            listObject.print();
242            break;
243      }
244   } while ( choice != 5 );
245
246   cout << "End list test\n\n";
247}
248
249void instructions()
250{
251   cout << "Enter one of the following:\n"
252        << "  1 to insert at beginning of list\n"
253        << "  2 to insert at end of list\n"
254        << "  3 to delete from beginning of list\n"
255        << "  4 to delete from end of list\n"
256        << "  5 to end list processing\n";
257}
258
```

Choices correspond to the switch statement

```
259int main()
260{
261   List< int > integerList;
262   testList( integerList, "integer" ); // test integerList
263
264   List< double > doubleList;
265   testList( doubleList, "double" );     // test doubleList
266
267   return 0;
268}
```

Use templates to create an integer list and a **`double`** list.

```
Testing a List of integer values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 1
Enter integer: 2
The list is: 2 1

? 2
Enter integer: 3
The list is: 2 1 3

? 2
Enter integer: 4
The list is: 2 1 3 4

? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4
```

```
? 4
4 removed from list
The list is: 3

? 4
3 removed from list
The list is empty

? 5
End list test

Testing a List of double values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter double: 1.1
The list is: 1.1

? 1
Enter double: 2.2
The list is: 2.2 1.1

? 2
Enter double: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter double: 4.4
The list is: 2.2 1.1 3.3 4.4
```

```
? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3

? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed
```

Class Templates and Non-type Parameters

- can use non-type parameters in templates
  - default argument
  - treated as **`const`**
- Example:

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

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

```
T stackHolder[ elements ]; //array to hold the list
```

    - creates array at compile time, rather than dynamic allocation at execution time

Templates and Inheritance

- A class template can be derived from a template class
- A class template can be derived from a non-template class
- A template class can be derived from a class template
- A non-template class can be derived from a class template

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