Unit 8 (Ch 17)

Templates

Prof. Chien-Nan (Jimmy) Liu
Dept. of Electronics & Electrical Engr.
Nat'l Yang Ming Chiao Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nycu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

- *8.1 Templates for Algorithm Abstraction*
  - Function Templates
- 8.2 Templates for Data Abstraction
  - Class Templates

Why Need Generic Template?

- Function definitions often use application specific adaptations of more general algorithms
- Ex: swapping integers and characters requires two different functions
  - Their codes are almost the same …
- Is it possible to have a generic algorithm?
  - Expressed independently of representation details

```
void swapValues(int& v1, int& v2)
{
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

```
void swapValues(char& v1, char& v2)
{
    char temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```


Templates for Functions

- How do we achieve that in C++ → function template
- Definition of function template
  - Argument types can also be PARAMETERS !!
- In this example, T is used as the type name
  - Could be used to swap values of any type accepted by C++
  - Automatically replace T with the type of given variable
  - An extra declaration, *template <class …>*, is required

```
template <class T>
void swapValues(T& v1, T& v2)
{
    T temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

```
void f() {
    int i1 = 10, i2 = 100;
    char c1 = 'O', c2 = 'X';
    string s1("abc"), s2("xyz");
    swap(i1, i2);    // call swap<int>(int&, int&)
    swap(c1, c2);    // call swap<char>(char&, char&)
    swap(s1, s2);    // call swap<string>(string&, string&)
}
```

Templates Prefix

- Template prefix:
  template <class T> or template <typename T>
  - Tells compiler that the declaration or definition that follows is a template
- Tells compiler that T serves as a type name
  - T is the traditional name, but can be any valid, non-keyword identifier, ex: template <class VariableType>

Type parameter

Template prefix →

```
template <class VariableType>
void swapValues(VariableType& v1, VariableType& v2)
{
    VariableType temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```


Templates Details

- A template overloads the function name by replacing T with the type used in a function call
  - Whether the type is a class or not
- There can be more than one template parameters and non-type parameters
  - Ex: template<class T1, class T2>
    - All parameters must be used in the template function
  - Ex: template<class T, int d2>
    - Give an extra argument while defining template functions
    - Treated as a constant

Calling a Template Function

- Calling a function defined with a template is identical to calling a normal function
  - Compiler checks the argument types and generates an appropriate version of the function
- Compiler only generates actual function definitions when required (i.e. instantiation-on-demand)
  - Similar to polymorphism??
- Ex: to call the template version of swapValues

```
char s1, s2;
int i1, i2;
…
swapValues(s1, s2);  → swapValue(char, char)
swapValues(i1, i2);  → swapValue(int, int)
```


Templates and Declarations

- You are not allowed to separate interface (header) and implementation files for template functions
  - At least the function declaration must precede any use of the template function
  - Need exact parameter type to match the implementation
- To be safe, you can
  - Place template function definitions in the same file where they are used (no separate file), or
  - Give the function template definition in one file and #include that file in another file that uses the template function, or
  - Put your definition and implementation, all in the header file
    - Include that header file will "copy" all code into the file that uses the template function

Algorithm Abstraction

- Algorithm abstraction allows us to express algorithms in a very general way
  - Use template functions in C++
- Help to concentrate on the substantive part of the algorithm
  - Allow us to ignore incidental detail
- Tips to write template function definition
  - Test the code with one type that might be needed
    - Easier to debug on the ordinary function
  - It could work for all types → generic programming

Ex: A Generic Sorting Function

- The sort function below uses an algorithm that does not depend on the base type of the array
  - Could be used to sort an array of any type
- This selection sort uses two helper functions
  - indexOfSmallest → find the smallest element
  - swapValues → exchange the locations of two elements
  - Both are defined with a template

```
void sort(BaseType a[], int numberUsed)
{
    int indexOfNextSmallest;
    for (int index = 0; index < numberUsed -1; index++)
    {
        indexOfNextSmallest =
                indexOfSmallest(a, index, numberUsed);
        swapValues(a[index], a[indexOfNextSmallest]);
    }
}
```

Templates and Operators

- The function indexOfSmallest compares items in an array using the < operator
  - If a class type is used in the template function, make sure you have the < operator overloaded for the class

```
int indexOfSmallest(const BaseType a[], int startIndex,
                                         int numberUsed)
{
    BaseType min = a[startIndex];
    int indexOfMin = startIndex;

    for (int index = startIndex+1; index < numberUsed; index++)
        if (a[index] < min) {
            min = a[index];
            indexOfMin = index;
        }
    return indexOfMin;
}
```


Code for Generic Sorting (1/2)

sortfunc.cpp

```
template<class T>
void swapValues(T& variable1, T& variable2)
{ /* as shown in previous slides */ }

template<class BaseType>
void sort(BaseType a[], int numberUsed)
{ /* as shown in previous slides */ }

template<class BaseType>
int indexOfSmallest(const BaseType a[],
             int startIndex, int numberUsed)
{ /* as shown in previous slides */ }
```

```
#include <iostream>
using namespace std;
#include "sortfunc.cpp"

int main( )
{
    int i;
    int a[10] = {9, 8, 7, 6, 5, 1, 2, 3, 0, 4};
    cout << "Unsorted integers:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;

    sort(a, 10);    // sort(int [], int)
    cout << "In sorted order the integers are:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;
```

Code for Generic Sorting (2/2)

```
double b[5] = {5.5, 4.4, 1.1, 3.3, 2.2};
cout << "Unsorted doubles:\n";
for (i = 0; i < 5; i++)
    cout << b[i] << " ";
cout << endl;

sort(b, 5);    // sort(double [], int)
cout << "In sorted order the doubles are:\n";
for (i = 0; i < 5; i++)
    cout << b[i] << " ";
cout << endl;

char c[7] = {'G', 'E', 'N', 'E', 'R', 'I', 'C'};
cout << "Unsorted characters:\n";
for (i = 0; i < 7; i++)
    cout << c[i] << " ";
cout << endl;
```

```
    sort(c, 7);    // sort(char [], int)
    cout << "In sorted order the characters are:\n";
    for (i = 0; i < 7; i++)
        cout << c[i] << " ";
    cout << endl;

    return 0;
}
```

Output

```
Unsorted integers:
9 8 7 6 5 1 2 3 0 4
In sorted order the integers are:
0 1 2 3 4 5 6 7 8 9
Unsorted doubles:
5.5 4.4 1.1 3.3 2.2
In sorted order the doubles are:
1.1 2.2 3.3 4.4 5.5
Unsorted characters:
G E N E R I C
In sorted order the characters are:
C E E G I N R
```


Function Template Overloading

- Like other functions, template functions can be overloaded, but quite confusing …

```
① template<class T> T sqrt(T);
② template<class T> complex<T> sqrt(complex<T>);
③ double sqrt(double);   // in <cmath>

   void f(complex<double> z) {
      sqrt(2);       // sqrt<int>(int) ①
      sqrt(2.0);     // double sqrt(double) ③
      sqrt(z);       // sqrt<double>(complex<double>) ②
   }
```

- General rules for resolving overloaded functions
  - Prefer ordinary functions to template functions
  - Consider only the most specialized template function

Inappropriate Types for Templates

- Supposedly, templates can be used for any type for which the code in the function makes sense
  - swapValues swaps individual objects of a type
- In fact, a function template usually imposes constraints for its use
  - Only works for those types with appropriate copy ctors and assignment operator
- This code would not work, because the assignment operator does not work with arrays:

```
int a[10], b[10];
<code to fill the arrays>
swapValues(a, b);
```


Overview

- 8.1 Templates for Algorithm Abstraction
  - Function Templates
- *8.2 Templates for Data Abstraction*
  - Class Templates

Templates for Data Abstraction

- Class definitions can also be general with templates
  - Syntax for class templates is almost the same
  - *template<class T>* comes before the class definition
  - Type parameter T is used in the class definition just like any other type
- This class contains a pair of values of type T

```
template <class T>
class Pair
{
  public:
     Pair( );
     Pair(T firstVal, T secondVal);
        …
     void setElement(int position,
                                    T value);
     T getElement(int position) const;
  private:            // position is 1 or 2
     T first;
     T second;
};
```


Template Class Objects

- Once the class template is defined, you can declare objects with any type
  - Must indicate what type is to be used for T
  - Ex: declare an object that holds a pair of integers:
    → Pair<int> score;
    or for a pair of characters: → Pair<char> seats;
- After declaration, objects based on a template class are used just like any other objects
  - Ex:

```
score.setElement(1,3);     // setElement(int, int)
score.setElement(2,0);     // setElement(int, int)
seats.setElement(1,'A');   // setElement(int, char)
```

Defining a Pair Constructor

- This is a definition of the constructor for class Pair that takes two arguments

```
template<class T>
Pair<T>::Pair(T firstVal, T secondVal)
          : first(firstVal), second(secondVal)
  {
    //No body needed due to initialization above
  }
```

  - The class name includes <T>

Defining the Member Functions

- Member functions of a template class are defined in the same way
  - The only difference is that the member function definitions also include templates (Pair → Pair<T>)
- Ex: definition of setElement in the template class Pair

```
void Pair<T>::setElement(int position, T value)
{
    if (position == 1)
      first = value;
    else if (position == 2)
      second = value;
    else
      …
}
```

Default Template Parameters (1/2)

```
template<class T = int, int size = 100>
class Array {
    T head[size];
  public:
    T& operator[](int idx);
    ……
};

template<class T, int size>   // don't redefine default value
T& Array<T, size>::operator[](int idx) {
    if (idx >= 0 && idx < size)  return head[idx];
    cerr << "Out-of-Range Error!\n";
    exit(1);
}
```

If no parameter is given, T is set to int

There is an extra non-type parameter, whose default value is 100

Default Template Parameters (2/2)

```
void f() {
    Array<int, 100> ia1;        // type = int, size = 100
    Array<> ia2;                // use default value, type = int, size = 100
    Array<char> ca;             // no size is given, type =char, size = 100
    ia1[20] = 101;              // use overloaded [] operator
    Array<int, 100> ia3(ia1);   // use default copy ctor
    ia2 = ia3;                  // use default assignment operator
    ca[ ia2[20] ] = 'X';        // cause a runtime range-checking error !!
};
```

Template Class Names as Parameters

- The name of a class can be used as the type of a function parameter

  → The name of a template class can be used as the type of a function parameter, too

  - Ex: create a parameter of type Pair<int>

```
int addUp(const Pair<int>& thePair);
//Returns the sum of two integers in thePair
```

- Function addUp can be made more general as a template function:

```
template<class T>
T addUp(const Pair<T>& thePair)
//Returns sum of the two values in thePair
```


Program Example: A Generic List

- The following example is a class template whose objects are lists
  - The lists can be lists of any type
- Not an actual link-list …
  - Use a dynamic array with *max* items to store data
    - The type of the items in this array is a template
  - One integer is used to keep the current length
  - Another is max length

```
template<class ItemType>
class GenericList
{
 public:
   GenericList(int max);
   ~GenericList();
   int length( ) const;
   void add(ItemType newItem);
   bool full( ) const;
   void erase( );   // remove all items from the list
   friend ostream& operator <<(ostream& outs,
          const GenericList<ItemType>& theList) {
     for (int i = 0; i < theList.currentLength; i++)
          outs << theList.item[i] << endl;
     return outs; }
 private:
   ItemType *item;
   int maxLength, currentLength;
};
```

Code for Generic List (1/2)

genericlist.h

```
#ifndef GENERICLIST_H
#define GENERICLIST_H
#include <iostream>
using namespace std;

namespace listsavitch
{ /* see previous slide */ }
#endif //GENERICLIST_H
```

test.cpp

```
#include <iostream>
#include "genericlist.h"
#include "genericlist.cpp"
using namespace std;
using namespace listsavitch;

int main( )
{
    GenericList<int> firstList(2);
    firstList.add(1);
    firstList.add(2);
    cout << "firstList = \n" << firstList;
    GenericList<char> secondList(10);
    secondList.add('A');
    secondList.add('B');
    secondList.add('C');

    cout << "secondList = \n" << secondList;
    return 0;
}
```

Output

```
first_list =
1
2
second_list =
A
B
C
```


Code for Generic List (2/2)

genericlist.cpp

```
#ifndef GENERICLIST_CPP
#define GENERICLIST_CPP
#include <cstdlib>
#include "genericlist.h"
using namespace std;

namespace listsavitch
{
    template<class ItemType>
    GenericList<ItemType>::GenericList(int max) :
              maxLength(max), currentLength(0) {
        item = new ItemType[max];
    }

    template<class ItemType>
    GenericList<ItemType>::~GenericList( ) {
        delete [] item; }
    }

    template<class ItemType>
    int GenericList<ItemType>::length( ) const {
        return (currentLength); }
    }

    template<class ItemType>
    void GenericList<ItemType>::
                    add(ItemType newItem)
    {
        if ( full( ) ) {
            cout << "Error: adding to a full list.\n";
            exit(1);
        }
        else {
            item[currentLength] = newItem;
            currentLength = currentLength + 1;
        }
    }

    template<class ItemType>
    bool GenericList<ItemType>::full( ) const {
        return (currentLength == maxLength);
    }

    template<class ItemType>
    void GenericList<ItemType>::erase( ) {
        currentLength = 0; }
} //listsavitch
#endif
```

Implementation Notes

- In the first version of class GenericList, we put the implementation of << operator in the header file
  - This is common for template classes with friend operators
  - Because << is not a member of the class, its implementation is simpler in this way
- We can put the implementation of << into the implementation file, but there is extra work
  - Make forward declaration with the diamond in the header file
  - Implement << in the .cpp file → see example as follows

Modified Header File for Generic List

```
#ifndef GENERICLIST_H
#define GENERICLIST_H
#include <iostream>
using namespace std;

namespace listsavitch
{
    template<class ItemType>
    class GenericList;

    template<class ItemType>
    ostream& operator <<(ostream& outs,
       const GenericList<ItemType>& theList);

    template<class ItemType>
    class GenericList
    {
     public:
       GenericList(int max);
       ~GenericList( );
       int length( ) const;
       void add(ItemType newItem);
       bool full( ) const;
       void erase( );
       friend ostream& operator << <>(ostream&
        outs, const GenericList<ItemType>& theList);

     private:
       ItemType *item;
        //pointer to the dynamic array for the list
       int maxLength;
        //max number of items allowed on the list.
       int currentLength;
        //number of items currently on the list.
    };

} //listsavitch
#endif //GENERICLIST_H
```

Modified genericlist.cpp

```
#ifndef GENERICLIST_CPP
#define GENERICLIST_CPP
#include <cstdlib>
#include "genericlist.h"
using namespace std;

namespace listsavitch {

   template<class ItemType>
   GenericList<ItemType>::GenericList(int max) :
              maxLength(max), currentLength(0)
   { /* the same as in previous example */ }

   template<class ItemType>
   GenericList<ItemType>::~GenericList( )
   { /* the same as in previous example */ }

   template<class ItemType>
   int GenericList<ItemType>::length( ) const
   { /* the same as in previous example */ }

   template<class ItemType>
   void GenericList<ItemType>::
                     add(ItemType newItem)
   { /* the same as in previous example */ }

   template<class ItemType>
   bool GenericList<ItemType>::full( ) const
   { /* the same as in previous example */ }

   template<class ItemType>
   void GenericList<ItemType>::erase( )
   { /* the same as in previous example */ }

   template<class ItemType>
   ostream& operator <<(ostream& outs,
       const GenericList<ItemType>& theList) {
     for (int i=0; i < theList.currentLength; i++)
         outs << theList.item[i] << endl;
     return outs;
   }
} //listsavitch
#endif
```


Friends and Templates

- With templates, there are different kinds of relationship between classes and their friends
- Many-to-one relationship: template function j() with any type is a friend to the regular non-template class B
  - j<int> and j<char> are both friends of class B
- One-to-many relationship: function e() is a friend to all instantiations of class A
  - e() is a friend of A<int> and A<char>
- One-to-one relationship:
  - g<int> is a friend of A<int>
  - f<char> is a friend of A<char>
  - f<int> is not a friend of A<char>

```
class B {
   template<class V> friend int j();
}

template<class S> g();

template<class T> class A {
   friend int e();
   friend int f(T);
   friend int g<T>();
};   // g must be defined first
```

typedef and Templates

- You specialize a class template by giving a type argument to the class name such as Pair<int>
  - The specialized name, Pair<int>, is used just like any class name
- You can define a new class type name with the same meaning as the specialized name:
  - typedef Class_Name<Type_Arg> New_Type_Name;
  - More convenient to use
  - For example:

```
typedef Pair<int> PairOfInt;
PairOfInt  pair1, pair2;  // equal to Pair<int> …
```


Summary

- Templates → type can be parameters
- Templates can enable
  - Generic programming
  - Containers (discussed in next chapter)
- Function templates
  - Template function overloading
  - Allow us to express algorithms in a very general way
- Class templates
  - Allow us to create similar objects with different types
- Watch for default template parameters, non-type parameters, and friendship

typedef and Templates

- You specialize a class template by giving a type argument to the class name such as Pair<int>
  - The specialized name, Pair<int>, is used just like any class name
- You can define a new class type name with the same meaning as the specialized name:
  - typedef Class_Name<Type_Arg> New_Type_Name;
  - More convenient to use
  - For example:

```
typedef Pair<int> PairOfInt;
PairOfInt  pair1, pair2;  // equal to Pair<int> …
```


Summary

- Templates → type can be parameters
- Templates can enable
  - Generic programming
  - Containers (discussed in next chapter)
- Function templates
  - Template function overloading
  - Allow us to express algorithms in a very general way
- Class templates
  - Allow us to create similar objects with different types
- Watch for default template parameters, non-type parameters, and friendship