Generic Programming: Template

C++ Object Oriented Programming

Pei-yih Ting

NTOU CS

Contents

Generic Functions

- Suppose all you want to do is copy an array regardless of type

```
void copy(int arrayTo[], int arrayFrom[], int n) {
    for (int i=0; i<n; i++)
        arrayTo[i] = arrayFrom[i];
}
```

The same function won't work on an array of doubles.

- A traditional solution

```
typedef int genericType;
void copy(genericType arrayTo[], genericType arrayFrom[], int n) {
    for (int i=0; i<n; i++)
        arrayTo[i] = arrayFrom[i];
}
```

This only works one type at a time.

- A C++ solution: overload the function

```
void copy(int arrayTo[], int arrayFrom[], int n);
void copy(double arrayTo[], double arrayFrom[], int n);
```

You still have to write separate functions for each type.
You have to know in advance what types you need.

Using Templates to Achieve Generality

- Template for toplevel functions

C++98/03

keyword only

parameter

or equivalently, **template <typename genericType>**

```
template<class genericType>
void copy(genericType arrayTo[], genericType arrayFrom[], int n) {
    for (int i=0; i<n; i++)
        arrayTo[i] = arrayFrom[i];
}
```

- Usage

```
void main() {
    int firstArray[] = {1, 2, 3};
    int secondArray[3];
    copy(secondArray, firstArray, 3);
}
```

or **copy<int>(secondArray, firstArray, 3)**

- What happens: the compiler instantiates the function with *int* as argument. If you call the same function with arrays of doubles, the compiler will instantiate a second function with *double* as argument.

Multiple Template Parameters

- A template parameter can not represent more than one type

```
template<class genericType>
void copy(genericType arrayTo[], genericType arrayFrom[], int n) {
   for (int i=0; i<n; i++)
      arrayTo[i] = arrayFrom[i];
}
void main() {
   int firstArray[] = {1, 2, 3};
   double secondArray[3];
   copy(secondArray, firstArray, 3);
}
```

```
error C2782: 'void __cdecl copy(genericType [],genericType [],int)' :
template parameter 'genericType' is ambiguous could be 'int' or 'double'
```

- The solution

```
copy<double,int>(secondArray, firstArray, 3)
```

```
template<class typeA, class typeB>
void copy(typeA arrayTo[], typeB arrayFrom[], int n) {
   for (int i=0; i<n; i++)
      arrayTo[i] = arrayFrom[i];
}
```

Template Errors

- You cannot violate syntax rules in a template

```
int array1[] = {1, 2, 3};
char *array2[3];
copy(array2, array1, 3);
```

```
error C2440: '=' : cannot convert from 'int' to 'char *'
```

- You cannot violate semantics

```
template<class type>
type add(type x, type y) {
   return x+y;
}
void main() {
   int int1=5, int2=6;
   double double1=7.2, double2=4.3;
   int array1[] = {1,2,3}, array2[] = {4,5,6};
   cout << add(int1, int2) << endl; // OK
   cout << add(double1, double2) << endl; // OK
   cout << add(int1, double2) << endl; // bad syntax
   cout << add(array1, array2) << endl; //error C2110: cannot add two pointers
}
```

```
error C2782: 'type __cdecl add(type,type)' :
        template parameter 'type' is ambiguous
        could be 'double' or 'int'
```

Improving the Semantics

```
template<class type>
type add(type x, type y) {
   return x+y;
}
class Array {
public:
   Array();
   void insert(int slot, double element);
   double get(int slot) const;
   void display() const;
   Array operator+(const Array &rhs) const;
private:
   double m_array[cArraySize];
};
void main() {
   Array array1, array2, array3;
   array1.insert(0, 2.2); array2.insert(0, 4.5);
   array3 = add(array1, array2);
   array3.display();
}
```

```
Array Array::operator+(const Array &rhs) const {
   Array tmp;
   for (int i=0; i<cArraySize; i++)
      tmp.m_array[i] = m_array[i] + rhs.m_array[i];
   return tmp;
}
```

```
Array::Array() {
   for (int i=0; i<cArraySize; i++)
      m_array[i] = 0;
}
```

```
Output
6.7 0 0
```

Templates and Overloading

- You can overload a template function

```
template<class type>
type add(type x, type y) {
   return x+y;
}
template<class type>
type add(type x, type y, type z) {
   return x+y+z;
}
void main() {
   int x = 5;
   int y = 4;
   int z = 1;
   cout << add(x, y) << endl;
   cout << add(x, y, z) << endl;
}
```

```
Output
9
10
```

- Overloading is more commonly used to *avoid* a template, see next page

Template and Overloading (cont'd)

- The template below will work fine with integers, doubles and chars

```
template <class type>
bool greaterThan(type x, type y) {
   return x > y;
}
```

But this template will fail with C char arrays.

- The solution is to provide an overloaded non-template function in addition to the template function

```
template <class type>
bool greaterThan(type x, type y) {
   return x > y;
}
bool greaterThan( char *str1, char *str2) {
   return strcmp(str1, str2) > 0;
}
```

- Rule for "signature matching" with templates: **non-tempalte functions have precedence over template functions in matching function calls**

Program Linkage Notes

- In a multi-file C++ project, we
  - put function prototypes in *.h file and put the definitions of each function in *.cpp files
  - put class declarations in *.h file and put the member function definitions in *.cpp files

  Which files should we put the template function into?

  *.cpp ? No. we should put template definitions into ***.h** file.
  - Remember that the compiler needs to instantiate the real function body according to the template function call statement. Therefore, the compiler need to know the complete template definitions before it can instantiate a templated function after seeing the function call statement.
  - Previously, the compiler only need to know the declaration of each class or function. The actual function codes are only required at linkage step.

Template Classes

- Template classes are the real reason templates were added to C++
- A complete array example

```
template <class type>
class Array {
public:
   Array(int arraySize);
   ~Array();
   void insert(int slot, type element);
   type get(int slot) const;
private:
   int m_arraySize;
   type *m_data;
};
template<class type>
Array<type>::Array(int arraySize): m_arraySize(arraySize) {
   m_data = new type[arraySize];
}
```

```
template<class type>
Array<type>::~Array() {
   delete[] m_data;
}
```

Template Classes (cont'd)

```
template<class type>
void Array<type>::insert(int slot, type element) {
   if (slot<m_arraySize && slot>=0)
      m_data[slot] = element;
   else
      cout << "Warning, out of range!\n";
}
template<class type>
type Array<type>::get(int slot) const {
 if (slot<m_arraySize && slot>=0)
      return m_data[slot];
   else
      cout << "Warning, out of range!\n";
   return 0; // return something
}
void main() {
   Array<int> array(20);
   array.insert(0, 10);
   cout << array.get(0);
}
```

You now have an array class that can hold chars, ints, doubles, strings, and other classes

Templates with Constant Parameters

- Templates can also include constant expressions

```
template<class type, int arraySize>
class Array {
public:
   void insert(int slot, type element);
   type get(int slot) const;
private:
   type m_data[arraySize];
};
```

- Usage

```
void main() {
   Array<int, 100> array;
   array.insert(99, 123);
   cout << array.get(99) << endl;
}
```

- Sample member function

```
template<class type, int arraySize>
void Array<type, arraySize>::insert(int slot, type element) {
   if (slot<arraySize && slot>=0)
      m_data[slot] = element;
   else
      cout << "Warning, out of range!\n";
}
```

Templates and Static Data Member

- When a template class contains a static data member, each instantiation type has its own static data member
- Consider this modification of the previous array template

```
template <class type>
class Array {
public:
   Array(int arraySize);
   ~Array();
   void insert(int slot, type element);
   type get(int slot) const;
private:
   int m_arraySize;
   type *m_data;
   static type sDefault;
};
```

- Every static data member must be *defined* outside the class
  In the case of templates, we can do this generically

```
template<class type>
type Array<type>::sDefault = 0;
```

Or on a type by type basis

```
int Array<int>::sDefault = 0;
char Array<char>::sDefault = '#';
```

Static Data Members (cont'd)

- The get() function returns the static data member

```
template<class type>
type Array<type>::get(int slot) const {
   if (slot<m_arraySize && slot>=0)
        return m_data[slot];
   else {
        cout << "Warning, out of range!\n";
        return sDefault; // return something
   }
}
```

- Usage

```
void main() {
   Array<char> array1(100);
   Array<int> array2(100);
   array1.insert(0, 'A');
   cout << array1.get(0) << endl;
   cout << array1.get(100) << endl; // out of range
   array2.insert(0, 5);
   cout << array2.get(0) << endl;
   cout << array2.get(100) << endl; // out of range
}
```

Output

```
A
Warning, out of range!
#
5
Warning, out of range!
0
```

Templated Classes with Templated Classes

- If an object within a templated class contains the parameterized variable, the object must also be a template
- Example (linked list)

```
template <class type>
class LinkedList {
public:
   LinkedList();
   ~LinkedList();
   void append(type value);
   void display();
private:
   Data<type> *m_tail;
   Data<type> *m_head;
};
```

```
template <class type>
class Data {
   friend class LinkedList<type>;
private:
   Data(type value);
private:
   type m_value;
   Data<type> *m_next;
};
```

```
template <class type>
Data<type>::Data(type value):m_value(value), m_next(0) {
}
```

Templates and Friends

```
template <class type>
LinkedList<type>::LinkedList(): m_head(0), m_tail(0) {
}
```

- Sample function from the linked list

```
template<class type>
void LinkedList<type>::append(type value) {
   Data<type> *tmp = new Data<type>(value);
   if (m_head == 0)
      m_head = tmp;
   else
      m_tail->m_next = tmp;
   m_tail = tmp;
}
```

- Usage

```
void main() {
   LinkedList<char> myLinkedList;
   myLinkedList.append('A');
}
```

Templated Member Function

```
//------ MyClass.h -----
class MyClass {
public:
   MyClass(void);
   template <class T> void func(T x);
};

#include <iostream>
#include <iomanip>
template <class T>
void MyClass::func(T x) {
   std::cout << x << std::endl;
}
//------ end of MyClass.h -----
```

or

```
template <class T> void func(T x) {
   std::cout << x << std::endl;
}
```

Design Considerations

- Usage for templates: primarily for container classes, i.e. arrays, stacks, linked lists, map, etc
  Commonly used in class libraries
- Example of template library
  STL (Standard Template Library) … The Standard C++ Library
- How to write a good template?
  Avoid including elements to the template that will defeat its generality.
- Examples:
  include a function that adds together two components
  Now you can’t use the template on any class that doesn’t overload+
- Document the template thoroughly.
  State which types will not work with the template.
  State which functions you expect to be available, e.g., +