Introduction to C++ Templates

Jonathan Hoyle

11/2/00

Overview

- What are templates?
- Templated Functions
- Templated Classes
- Templated Methods
- Advanced Topics
- Coding Examples

What are templates?

- Functions or classes with type as a parameter
- Angle brackets < > are used to specify the type variable:

```
MyStndClass          theStndClass;
MyTmplClass<int>     theTmplClass;
```

- Template declarations begin like this:

```
template <class T>
```

Example

```
void Clamp(int &x, int min, int max)
{ if (x < min) x = min; if (x > max) x = max; }

void Clamp(short &x, short min, short max)
{ if (x < min) x = min; if (x > max) x = max; }

void Clamp(long &x, long min, long max)
{ if (x < min) x = min; if (x > max) x = max; }

void Clamp(double &x, double min, double max)
{ if (x < min) x = min; if (x > max) x = max; }
```

Templated Functions

- Using templates, we can reduce this:

```
// templated definition for Clamp()
template <class T>
void Clamp(T &x, T min, T max)
{ if (x < min) x = min; if (x > max) x = max; }

Clamp(myInt, 0, 255);           // calls Clamp<int>
Clamp(myDouble, 0.0, 1.0);      // calls Clamp<double>
Clamp(myLong, 0L, 65535L);      // calls Clamp<long>
Clamp(myChar, 'A', 'Z'); // calls Clamp<char>
```

Template Specialization

- Sometimes you want to specialize your templated function depending on type:

```
// templated definition for Clamp()
template <class T>
void Clamp(T &x, T min, T max)
{ if (x < min) x = min; if (x > max) x = max; }

// char specialization for Clamp()
void Clamp(char &x, char min, char max)
{ if (x < min) x = ' '; if (x > max) x = ' '; }
```

Multiple Template Parameters

- You can vary on more than one type:

```
template <class T1, class T2, class T3>
void Clamp(T1 &x, T2 min, T3 max)
{if (x < min) x = (T1)min; if (x > max) x = (T1)max;}

Clamp(inLong, 0, 4.0);  // Clamp<long, int, double>

template <class T2, class T3>
void Clamp(char &x, T2 min, T3 max)
{ if (x < min) x = ' '; if (x > max) x = ' '; }

Clamp(inChar, 0, 127.0); // Clamp<char, int, double>
```

Templated Classes

- Classes (and structs) can be templated:

```
template <class T>
class Array
{
  public:
      ...
  protected:
      T     mArray[MAX_SIZE];
};

Array              myArray;    // Compiler ERROR
Array<int>         myIntArray; // array of int's
Array<void *>      myPtrArray; // array of ptr's
```

Template Parameters

- Template parameters can be non-types:

```
template <class T, int maxSize>
class Array
{
  public:
     ...
  protected:
     T     mArray[maxSize];
};

Array<int, 1024>  myIntArray;   // 1024 int's
Array<long, 1024> myLongArray1; // 1024 long's
Array<long, 256>  myLongArray2; // 256 long's
```

Template Parameter Defaults

- Template parameters can have defaults:

```
template <class T = int, int maxSize = 1024>
class Array
{
  public:
      ...
  protected:
      T     mArray[maxSize];
};

Array<>             myIntArray;   // 1024 int's
Array<long>         myLongArray1; // 1024 long's
Array<long, 256>    myLongArray2; // 256 long's
```

Template Class Method Definition

```
// Templated Foo< > declaration
template <class T>
class Foo
{
   public:
      ...
      T bar();
};

// Definition of bar() method in Foo<>
template <class T>
T Foo<T>::bar()
{ … }
```

Templated Methods

- Classes can have templated methods:

```
class DebugDialog
{
  public:
      ...
      template <class T>
      void displayToUser(T input)
      { cout << input << endl; }
};

dd.displayToUser("Hello, World!");
dd.displayToUser(theErrorNum);
dd.displayToUser(myInvalidPtr);
```

Template Gotcha's

- Both the definition *and implementation* must be #included in a header file
- Code is usually inlined for speed
- Compiler does not generate code unless a call is made to that particular method
- Syntax is not checked unless a call is made
- Templated classes of different types are treated as distinct types
- Can't be used with Visual C++ v1.5.2

Advanced Template Issues

- Explicit template definition creates code:

```
template <class T>
class Foo { … };

class Foo<int>;   // explicit definition

Foo<double> f1;   // inlined code
Foo<int>    f2;   // pre-created code
```

- Templates of templates: be careful of “>>”:

```
A<B<int>>   myVar;      // Compiler ERROR
A< B<int> > myVar;      // OK
```

Advanced Template Issues

- Ambiguous template functions:

```
template <class T>
void Clamp(T &x, T min, T max)
{ if (x < min) x = min; if (x > max) x = max; }

Clamp(myLong, 0, 256);          // Compiler ERROR
Clamp<long>(myLong, 0, 256);    // OK
```

- Template class methods cannot be virtual:

```
template <class T> class Foo
{
        void bar1();            // OK
        virtual void bar2();    // ERROR
};
```

Advanced Template Issues

- Non-type template parameters must be integral, pointer or reference:

```
template <class T, void *thePtr>
class Foo1 { … };  // OK

template <class T, double theValue>
class Foo2 { … };  // Compiler ERROR

template <class T, double &theValue>
class Foo3 { … };  // OK

template <class T, T *myClassPtr>
class Foo4 { … };  // OK
```

Advanced Template Issues

- T vs. const T& in method prototypes:

```
template <class T>
class Foo
{
        void bar1(T inT);
        void bar2(const T &inT);
              };

Foo<int>     f;
int          x = 0;

f.bar1(x);  // less optimal than bar2()
f.bar2(0);  // ERROR: can't reference 0
```

Template Resources

- Chapter 14 of the ANSI C++ Specification
- Chapter 7 of *The C++ Primer*, Lippman
- ANSI's STL (Standard Template Library)
- Microsoft's ATL
- A number of third party frameworks

Demo