5/18/2011

Template Specialization and Overloading

Klaus Fritzsche

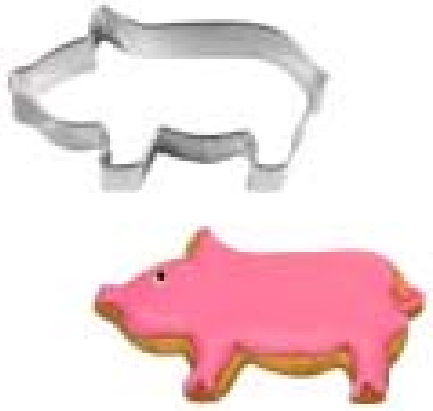
dkfz.
GERMAN
CANCER RESEARCH CENTER
IN THE HELMHOLTZ ASSOCIATION

Outline

1. What is template specialization? How do I do it?
2. What is partial specialization? How do I do it?
3. How do I determine which template gets called?

What is template specialization?

- Templates
  - C++'s most powerful form of genericity
- Code becomes
  - generic
  - data type independent



What is template specialization?

- Template specialization
  - Special treatment of certain types
  - Example: Fast version for char-array, slower but more generic approach for all other cases
  - Example: special version for objects that don't conform to the normal interface expected by the generic template

- Two forms of template specialization:
  - Explicit specialization
  - Partial specialization

Explicit Specialization

- Specific implementation for a particular combination of template parameters

- Example: explicitly specialization for char*'s

```
template<class T> void sort(Array<T>& v) { /*...*/ };

template<> void sort<char*>(Array<char*>&);
```

- The compiler will choose the most appropriate template:

```
Array<int> ai; Array<char*> apc;
sort( ai );  // calls sort<int>
sort( apc ); // calls specialized sort<char*>
```

Partial Specialization (1)

- For class templates only
- Don't fix all of the primary class template's parameters

```
// #1    general case:
template<class T1, class T2, int I>
        class A { };

// #2    T2 is a T1*:
template<class T, int I>
        class A<T, T*, I> { };

// #3    T1 is any pointer:
template<class T1, class T2, int I>
        class A<T1*, T2, I>{};
```

Partial Specialization (2)

- For class templates only
- Don't fix all of the primary class template's parameters

```
// #1    general case:
template<class T1, class T2, int I>
        class A { };

// #4    T1 is int and T2 is any pointer and I is 5:
template<class T>
        class A<int, T*, 5> { };

// #5    T2 is any pointer:
template<class T1, class T2, int I>
        class A<T1, T2*, I> { };
```

Partial Specialization (3)

```
// #1    general case:
template<class T1, class T2, int I>  class A { };

// #2    T2 is a T1*:
template<class T, int I>             class A<T, T*, I> { };

// #3    T1 is any pointer:
template<class T1, class T2, int I>  class A<T1*, T2, I>{};

// #4    T1 is int and T2 is any pointer and I is 5:
template<class T>                    class A<int, T*, 5> { };

// #5    T2 is any pointer:
template<class T1, class T2, int I>  class A<T1, T2*, I> { };
```

```
A<int, int, 1> a1;       // uses #1
```

Partial Specialization (4)

```
// #1    general case:
template<class T1, class T2, int I>  class A { };

// #2    T2 is a T1*:
template<class T, int I>             class A<T, T*, I> { };

// #3    T1 is any pointer:
template<class T1, class T2, int I>  class A<T1*, T2, I>{};

// #4    T1 is int and T2 is any pointer and I is 5:
template<class T>                    class A<int, T*, 5> { };

// #5    T2 is any pointer:
template<class T1, class T2, int I>  class A<T1, T2*, I> { };
```

```
A<int, int*, 1> a2;    // uses #2, T is int,
                       // I is 1
```

Partial Specialization (5)

```
// #1    general case:
template<class T1, class T2, int I>  class A { };

// #2    T2 is a T1*:
template<class T, int I>             class A<T, T*, I> { };

// #3    T1 is any pointer:
template<class T1, class T2, int I>  class A<T1*, T2, I>{};

// #4    T1 is int and T2 is any pointer and I is 5:
template<class T>                    class A<int, T*, 5> { };

// #5    T2 is any pointer:
template<class T1, class T2, int I>  class A<T1, T2*, I> { };
```

```
A<int, char*, 5> a3;  // uses #4, T is char
```

Partial Specialization (6)

```
// #1    general case:
template<class T1, class T2, int I>  class A { };

// #2    T2 is a T1*:
template<class T, int I>             class A<T, T*, I> { };

// #3    T1 is any pointer:
template<class T1, class T2, int I>  class A<T1*, T2, I>{};

// #4    T1 is int and T2 is any pointer and I is 5:
template<class T>                    class A<int, T*, 5> { };

// #5    T2 is any pointer:
template<class T1, class T2, int I>  class A<T1, T2*, I> { };
```

```
A<int, char*, 1> a4;  // uses #5, T1 is int,
                      // T2 is char, I is 1
```

Partial Specialization (7)

```
// #1    general case:
template<class T1, class T2, int I>  class A { };

// #2    T2 is a T1*:
template<class T, int I>             class A<T, T*, I> { };

// #3    T1 is any pointer:
template<class T1, class T2, int I>  class A<T1*, T2, I>{};

// #4    T1 is int and T2 is any pointer and I is 5:
template<class T>                    class A<int, T*, 5> { };

// #5    T2 is any pointer:
template<class T1, class T2, int I>  class A<T1, T2*, I> { };
```

```
A<int*, int*, 2> a5;  // ambiguous:
                      // matches #3 and #5
```

Function Template Overloading (1)

- C++ allows function overloading:

```
int f( int );
long f( double );

int i;
double d;

f( i );      // calls f(int)
f( d );      // calls f(double)
```

Function Template Overloading (2)

- Similarly, function templates can be overloaded

```
template<typename T1, typename T2> void f( T1, T2 ); // 1
template<typename T> void f( T, T );      // 2
template<typename T> void f( T*, T );     // 3
template<typename T> void f( T, T* );     // 4
template<typename T> void f( int, T* );   // 5
void f( int, double );                    // 6

template<typename T> void f( T );         // 7
template<typename T> void f( T* );        // 8
template<> void f<int>( int );            // 9
void f( int );                            // 10
```

Function Template Overloading (3)

```
template<typename T1, typename T2> void f( T1, T2 ); // 1
template<typename T> void f( T, T );      // 2
template<typename T> void f( T*, T );     // 3
template<typename T> void f( T, T* );     // 4
template<typename T> void f( int, T* );   // 5
void f( int, double );                    // 6

template<typename T> void f( T );         // 7
template<typename T> void f( T* );        // 8
template<> void f<int>( int );            // 9
void f( int );                            // 10
```

```
int i; double d; float ff; complex<double> c;
f( i );         // calls #10, exact match, non-templates
                  always preferred over templates
f<int>( i );    // calls #9, f<int> explicitly requested
f( i, i );      // calls #2, best match
f( c );         // calls #7, no other f matches
```

Function Template Overloading (4)

```
template<typename T1, typename T2> void f( T1, T2 ); // 1
template<typename T> void f( T, T );      // 2
template<typename T> void f( T*, T );     // 3
template<typename T> void f( T, T* );     // 4
template<typename T> void f( int, T* );   // 5
void f( int, double );                    // 6

template<typename T> void f( T );         // 7
template<typename T> void f( T* );        // 8
template<> void f<int>( int );            // 9
void f( int );                            // 10
```

```
int i; double d; float ff; complex<double> c;
f( i, ff ); // calls #1, #6 is very close, but non-
            templates preferred only for exact matches
f( i, d );  // calls #6, exact match nontemplate preferred
f( c, &c ); // calls #4, second parameter is a pointer to
            the same type as the first parameter
```

Function Template Overloading (5)

```
template<typename T1, typename T2> void f( T1, T2 ); // 1
template<typename T> void f( T, T );      // 2
template<typename T> void f( T*, T );     // 3
template<typename T> void f( T, T* );     // 4
template<typename T> void f( int, T* );   // 5
void f( int, double );                    // 6

template<typename T> void f( T );         // 7
template<typename T> void f( T* );        // 8
template<> void f<int>( int );            // 9
void f( int );                            // 10
```

```
int i; double d; float ff; complex<double> c;
f( i, &d );  // calls #5, closest overload
f( &d, d );  // calls #3, first parameter is a pointer to
                the same type as the second parameter
f( &d );     // calls #8
```

Function Template Overloading (6)

```
template<typename T1, typename T2> void f( T1, T2 ); // 1
template<typename T> void f( T, T );      // 2
template<typename T> void f( T*, T );     // 3
template<typename T> void f( T, T* );     // 4
template<typename T> void f( int, T* );   // 5
void f( int, double );                    // 6

template<typename T> void f( T );         // 7
template<typename T> void f( T* );        // 8
template<> void f<int>( int );            // 9
void f( int );                            // 10
```

```
int i; double d; float ff; complex<double> c;
f( d, &i );   // Several other overloads are close, but
                only #1 fits best
f( &i, &i ); // calls #2, closest overload even though
                some of the others explicitly mention a
                pointer parameter.
```

THANK YOU

Thanks also to Herb Sutter (www.GotW.ca), who inspired most of the slides