Template Specialization

Specialization

- When we define a template (class or function), our intent is to define a set of behaviors that is common across types.
- That's why we call this type independent or generic code.
- However, its certainly possible the generalization does not fit ALL types.
- And if this is the case, then you would want to customize that template for such type(s).
- This means you are specializing the template for such type(s).

Specialization

- And it also means that the specialized version of your template will be instantiated for these types, and not the primary template.

- Which means the compiler will match the specialization (if there is one) before the primary template (of course, as long as the types at usage point match exactly with the ones specified in the specialization).

Explicit (full) or partial specialization

- Now, if for a given specialization of your template, only the specific type(s) can be used (for which it is specialized), and you cannot pass in any other type, then it is explicit (full , total) specialization.

- If the specialization is such that you did not specify all type arguments in your specialization, then it means some general types can still be passed in to this. In this case, it is a partial specialization.

Full or partial specialization

- Lets say your template class takes two type parameters, T1 and T2.
- You could provide a specialization for only one of the types, and leave the other type as a parameter to be passed in at instantiation time.
- This is **partial** specialization.
- OR
-
- You could provide a specialized class where you specify both the types(say, string and int).
- This is **explicit(total**) specialization, because you specified another class (specialized version) which would be called when you instantiate the template class with string and int.

Specialization

```
template< typename T1, typename T2 >
class MyClass {                    // no specialization. You have to pass in T1 and T2
  // ...
}
```

This will be used as:

```
void Foo()
{
    MyClass < long, double > mcld;   // assume there is a default constructor

}
```

Partial specialization

- Below is partial specialization number 1.

- The second type parameter is int.

```
template< typename T1>
class MyClass < T1, int > { … }          // T2 is bound to int.
```

- This will be used as:

```
MyClass < double, int> mcdi;          // assume there is a default constructor.
```

- When the compiler sees the line above, it will match it to partial specialization #1 above.

- Even though the template class declared first (with no specialization) will also match this, but the rule is to take the **most specialized version** available.

Partial specialization

- Below is partial specialization number 2.
- The first type parameter is int.

```
template< typename T2>
class MyClass< int, T2 > { … }          // T1 is bound to int.
```

- This will be used as:
- MyClass < int, double > mcid;

Partial specialization

- Below is partial specialization number 3.
- The first type parameter is string.

```
template< typename T2>
class MyClass< string, T2 > { … }     // T1 is bound to string.
```

- This will be used as:

```
MyClass < string, double > mcsd;
```

Total specialization

- Below is a **total** specialization.

```
template < >          // Cannot pass in any type. This one is totally specialized.
class MyClass < string, string > { … }    // T1 and T2 are bound to string.
```

- This will be used as:

```
MyClass< string, string > mcss;
```

Specialization

- So, lets see some more declarations here and see what template gets instantiated as a result.
-
- MyClass <long, long> mcll; // uses the no specialization version
- MyClass <MyClass2, MyClass3> mc23; // uses the no specialization version
- MyClass < MyClass2, int> mc2i; // uses partial specialization #1
- MyClass < int, MyClass3 > mci3; // uses partial specialization #2
- MyClass < string, MyClass3 > mcs3; // uses partial specialization #3
- MyClass < string, string > mcss; // uses total specialization

- Keep in mind that it is not common to provide so many specializations.
- I have shown these here to get the point across.

Specialization

- When you specialize your template class, you do not have to maintain the same interface as the primary template class.
- You can have more methods or less methods, and also different data members in the specialized versions of the template class.

Partial specialization

- Lets say there is a class

template < typename T >
class Container { .. }

And we provide a specialization for pointers:

template <>
class Container < T * > { … }

This is a partial template specialization, and will be used for a container of pointers.

Now, we can use a technique similar to the template hoisting that we had seen in the template class.

That technique was used for saving code bloat, because we were deriving our template class from a non-template class.

Partial specialization

- So, in this case, we could do something like:

```
template <>
class Container < void * > { .. }
```

This is a total specialization for a vector of void *

Now, we could implement a vector of pointers in terms of a vector of void *
Since vector of void * is a total specialization, there is no type parameter involved in it, and hence there will be only one copy of its code, and not one copy for each different type of instantiation (same technique/reasoning as in template hoisting).

```
template < typename T>
class Container < T *>:  private  Container< void * >
{ .. }
```

Partial specialization

- Now, what we are saying is that all container of pointers are implemented in terms of Container <void *>

- And this will reduce the amount of code instantiated.

Specialization

- So, the specialization can be for one or both of the following reasons:
- Interface specialization
- Implementation specialization

vector < bool >

- vector < bool > is a specialization (partial) for bool.

- Primary template for vector stores the elements in an array of T.
- vector of bool stores integers and uses its bits to represent the bool elements.

- It has a slightly different interface from the primary vector template:

  - Method flip to flip the bits
  - Additional flavor of swap to swap two bool elements.

Explicit instantiation

- Note: This is not common.
- template class MyClass < double >;
- Notice there are no angle brackets after keyword template.
- This is an explicit instantiation of the template for type double.
- ALL members of the template will be instantiated, unlike implicit instantiation where only the members that are called are instantiated.
- Implicit instantiation is the one most commonly used.

Declaring and defining…

- You cannot /should not define a specialization for a type T *after* the use of T in the primary template.

- Example:

```
template < typename T > class MyClass { // … };

void Foo()
{
        MyClass < string >          mcStr;
        // …
}

template <> class MyClass < string > { //… };
```

This specialization is defined after MyClass < string > was used (instantiated) above. (See code example)

Declaring and defining…

template <> class MyClass < ABC > { //… };

// This is fine, since MyClass <ABC> has <u>not</u> been used (instantiated) so far

- In general, it is safest to <u>declare</u> the primary template and its specializations <u>before</u> you <u>define</u> them.
- That way, there is no chance that something would get instantiated before you actually declared /defined it.

Function Specialization

- You can also specialize function templates.

- You can say for a type, say, int, you want the specialized version of the template function to be called, and for any other type, call the non-specialized function template.

- Example:

```
template <typename T >
void Foo( T a );
```

- Below is a total specialization for int.

```
template <>
void  Foo <int> ( int a );
```

Function Specialization

Below is a total specialization for string.

```
template <>
void Foo <string> ( string a );
```

Function Specialization

- Note that you cannot do partial specialization for function templates.

- You have to do full (explicit) specialization.

- Now, C++ also allows overload of functions.
- And you can do this with template functions as well.
- This is different from template specialization.
-
- Below is an overload of function Foo for pointers.

```
template <typename T >
void Foo( T * a );
```

Function Specialization

- Below is another overload that takes two pointer parameters.

```
template <typename T >
void Foo( T * a, T * b );
```

Function Specialization

- If you really want partial specialization for functions, you can do something like the following (since partial specialization is not allowed on functions):

```
template <typename T>
class FuncSpecializer {
public:
        static void  FuncFoo();
};
template <typename T>
class FuncSpecializer < T * > {          // partial specialization of class template
public:
        static void  FuncFoo();
};

FuncSpecializer < MyClass * > ::FuncFoo();   // this will invoke FuncFoo
FuncSpecializer < int * > ::FuncFoo();         // this will invoke FuncFoo
```

Declaring and defining…

- Similar to what we saw for class templates, you should declare the primary function template and any overloads / specializations before you define them.

- This will ensure that the right specializations / overloads will get used.

- Declarations:

```
template < typename T >  void  FuncFoo( const T & ) ;  // primary template
template <>   void FuncFoo < string > ( const string & ); // specialization
```

Definitions:

```
template < typename T >  void  FuncFoo( const T & )   // primary template
{ … }

template <>   void FuncFoo < string > ( const string & ) // specialization
{ … }
```