Deep Down Boostimagical Fun

`boost::enable_if`

Christian Bay and Kasper A Andersen

Department of Computer Science

University of Copenhagen

SFINAE revisited

What is SFINAE?
Easy: "Substitution Failure Is Not An Error"

Example:

```
void dodah( int i ) // Version 1
{
    cout << "I: " << i << endl;
}

template <typename T>
void dodah( T i ) // Version 2
{
    typename T::result_type result_type;
    cout << "I: " << i << endl;
}
```

SFINAE revisited (contd.)

What happens now?

```
int main()
{
    int i;
    some_class_with_result_type c;
    short s;

    dodah( i ); // Version 1 called
    dodah( c ); // Version 2 called
    dodah( s ); // ???

    return 0;
}
```

SFINAE revisited (contd.)

Visual Studio is not really happy:

```
Examples.h(17) : error C2653: 'T' : is not a class or namespace
name; main.cpp(13) : see reference to function template
instantiation 'void dodah<short>(C) with [C=short]' being
compiled

Examples.h(17) : error C2065: 'result_type' : undeclared
identifier

Examples.h(17) : error C2146: syntax error : missing ';'
before identifier 'result_type'

Examples.h(17) : error C3861: 'result_type': identifier not
found, even with argument-dependent lookup
```

SFINAE revisited (contd.)

By reorganizing a little, SFINAE kicks in:

```
template <typename T>
typename T::result_type* dodah( T i )
{
    cout << "I: " << i << endl;
    return 0;
}

// Alternatively
template <typename T>
void dodah( T i, typename T::result_type* p=0 )
{
    cout << "I: " << i << endl;
}
```

Using enable_if

Let's try to use `enable_if`:

```
template <typename T>
void dodah( T i, typename disable_if<is_integral<T> >::type* p=0 )
{
    cout << "I: " << i << endl;
}
```

This is our new main template.
All integral types goes to version 1.

Using enable_if (contd.)

What about classes?

```
template <typename T, typename Enable = void>
class Dodah
{
    // General implementation
};

template <typename T>
class Dodah <T, typename enable_if<is_integral<T> >::type>
{
    // Special implementation for integral types
};
```

Implementation of enable_if

```
template <bool B, class T = void>
struct enable_if_c
{ typedef T type; }

template <class T>
struct enable_if_c<false, T>
{}

// Wrapper extracting 'value' from 'Cond'
template <class Cond, class T = void>
struct enable_if : public enable_if_c<Cond::value, T>
{}
```

Similarly, we have `disable_if`.
And then we have the lazy cousin...

Lazy enable_if

Remember lazy eveluation?
`enable_if` has something similar:

```
template <bool B, class T>
struct lazy_enable_if_c
{ typedef typename T::type type; }

template <class T>
struct lazy_enable_if_c<false, T>
{ // ::type is not extracted if condition is false };

template <class Cond, class T>
struct lazy_enable_if : public lazy_enable_if_c<Cond::value, T>
{};
```

Using the lazy version

```
template <typename T> typename enable_if<
    is_integral<T>, typename int_traits<T>::type>::type
dodah( T& const t )
{ // Implementation }

template <typename T> typename lazy_enable_if<
    is_integral<T>, int_traits<T> >::type
dodah( T& const t )
{ // Implementation }
```

The first one fails if `int_traits<T>::type` is not defined even though `is_integral<T>` is false. SFINAE is not used since invalid type occurs as an argument to another template!

Not portable

Not all compilers support SFINAE. Boost has a fine macro that helps disabling boost `enable_if` on those compilers.

```
// Even the definition of enable_if causes problems on some compilers,
// so it's macroed out for all compilers that do not support SFINAE

#ifndef BOOST_NO_SFINAE
   ...
#else
  ...

  template <class Cond, class T = detail::enable_if_default_T>
  struct enable_if : enable_if_does_not_work_on_this_compiler<T>
  { };

  ...
#endif // BOOST_NO_SFINAE
```

Not portable (cont.)

A nice error message instead?

```
error C2504:
'boost::enable_if_does_not_work_on_this_compiler<T> with
[T=boost::detail::enable_if_default_T]' : base class undefined;
see reference to class template instantiation
'boost::enable_if_does_not_work_on_this_compiler<T> with
[T=boost::detail::enable_if_default_T]' being compiled;
...
```

Our conclusion: Rethink your design if you really need `enable_if`. Use different function names, full template specializations (that calls the right specialized function to minimize the code that needs to be maintained).