More C++ Idioms/enable-if

Contents

enable-if

Intent

Allow function overloading based on arbitrary properties of type

Also Known As

- Explicit Overload Set Management

Motivation

The **enable_if** family of templates is a set of tools to allow a function template or a class template specialization to include or exclude itself from a set of matching functions or specializations based on properties of its template arguments. For example, one can define function templates that are only enabled for, and thus only match, an arbitrary set of types defined by a traits class. The **enable_if** templates can also be applied to enable class template specializations. Applications of **enable_if** are discussed in length in the literature.[1] [2]

Sensible operation of template function overloading in C++ relies on the SFINAE (substitution-failure-is-not-an-error) principle:[3] if an invalid argument or return type is formed during the instantiation of a function template, the instantiation is removed from the overload resolution set

instead of causing a compilation error. The following example[1] demonstrates why this is important:

```
int negate(int i) { return -i; }

template <class F>
typename F::result_type negate(const F& f) { return -f(); }
```

Suppose the compiler encounters the call *negate(1)*. The first definition is obviously a better match, but the compiler must nevertheless consider (and instantiate the prototypes) of both definitions to find this out. Instantiating the latter definition with *F* as *int* would result in:

```
int::result_type negate(const int&);
```

where the return type is invalid. If this was an error, adding an unrelated function template (that was never called) could break otherwise valid code. Due to the SFINAE principle the above example is not, however, erroneous. The latter definition of negate is simply removed from the overload resolution set.

The **enable_if** templates are tools for controlled creation of the SFINAE conditions.

Solution and Sample code

The enable_if templates are very simple syntactically. They always come in pairs: one of them is empty and the other one has a `type` typedef that forwards its second type parameter. The empty structure triggers an invalid type because it contains no member. When a compile-time condition is false, the empty `enable_if` template is chosen. Appending `::type` would result in an invalid instantiation, which the compiler throws away due to the SFINAE principle.

```
template <bool, class T = void>
struct enable_if
{};

template <class T>
struct enable_if<true, T>
{
  typedef T type;
};
```

Here is an example that shows how an overloaded template function can be selected at compile-time based on arbitrary properties of the type parameter. Imagine that the function **T foo(T t)** is defined for all types such that T is arithmetic. The enable_if template can be used either as the return type, as in this example:

```
template <class T>
typename enable_if<is_arithmetic<T>::value, T>::type
foo(T t)
{
```

```
  // ...
  return t;
}
```

or as an extra argument, as in the following:

```
template <class T>
T foo(T t, typename enable_if<is_arithmetic<T>::value >::type* dummy = 0);
```

The extra argument added to foo() is given a default value. Since the caller of foo() will ignore this dummy argument, it can be given any type. In particular, we can allow it to be void *. With this in mind, we can simply omit the second template argument to **enable_if**, which means that the **enable_if<...>::type** expression will evaluate to *void* when **is_arithmetic<T>** is true.

Whether to write the enabler as an argument or within the return type is largely a matter of taste, but for certain functions, only one alternative is possible:

- Operators have a fixed number of arguments, thus **enable_if** must be used in the return type.
- Constructors and destructors do not have a return type; an extra argument is the only option.
- There does not seem to be a way to specify an enabler for a conversion operator. Converting constructors, however, can have enablers as extra default arguments.

Known Uses

Boost library, C++ STL, etc.

Related Idioms

- SFINAE

References

1. Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, and Andrew Lumsdaine. Function overloading based on arbitrary properties of types. C/C++ Users Journal, 21(6):25--32, June 2003.
2. Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-controlled polymorphism. In Frank Pfennig and Yannis Smaragdakis, editors, Generative Programming and Component Engineering, volume 2830 of LNCS, pages 228--244. Springer Verlag, September 2003.

3. David Vandevoorde and Nicolai M. Josuttis. C++ Templates: The Complete Guide. Addison-Wesley, 2002.

External links

Boost Libraries enable-if documentation (http://www.boost.org/libs/utility/enable_if.html)
