Chapter 49. Boost.EnableIf

Boost.EnableIf makes it possible to disable overloaded function templates or specialized class templates. Disabling means that the compiler ignores the respective templates. This helps to prevent ambiguous scenarios in which the compiler doesn't know which overloaded function template to use. It also makes it easier to define templates that can be used not just for a certain type but for a group of types.

Since C++11, Boost.EnableIf has been part of the standard library. You can call the functions introduced in this chapter without using a Boost library; just include the header file `type_traits`.

Example 49.1. Overloading functions with `boost::enable_if` on their return value

```cpp
#include <boost/utility/enable_if.hpp>
#include <type_traits>
#include <string>
#include <iostream>

template <typename T>
typename boost::enable_if<std::is_same<T, int>, T>::type create()
{
  return 1;
}

template <typename T>
typename boost::enable_if<std::is_same<T, std::string>, T>::type create()
{
  return "Boost";
}

int main()
{
  std::cout << create<std::string>() << '\n';
}
```

Example 49.1 defines the function template `create()`, which returns an object of the type passed as a template parameter. The object is initialized in `create()`, which accepts no parameters. The signatures of the two `create()` functions don't differ. In that respect `create()` isn't an overloaded function. The compiler would report an error if Boost.EnableIf didn't enable one function and disable the other.

Boost.EnableIf provides the class `boost::enable_if`, which is a template that expects two parameters. The first parameter is a condition. The second parameter is the type of the `boost::enable_if` expression if the condition is true. The trick is that this type doesn't exist if the condition is false, in which case the `boost::enable_if` expression is invalid C++ code. However, when it comes to templates, the compiler doesn't complain about invalid code. Instead it ignores the template and searches for another one that might fit. This concept is known as SFINAE which stands for "Substitution Failure Is Not An Error."

In Example 49.1 both conditions in the `boost::enable_if` expressions use the class `std::is_same`. This class is defined in the C++11 standard library and allows you to compare two types. Because such a comparison is either true or false, it's sufficient to use `std::is_same` to define a condition.

If a condition is true, the respective `create()` function should return an object of the type that was passed to `create()` as a template parameter. That's why `T` is passed as a second parameter to `boost::enable_if`. The entire `boost::enable_if` expression is replaced by `T` if the condition is true. In Example 49.1 the compiler sees either a function that returns an `int` or a function that returns a `std::string`. If `create()` is called with any other type than `int` or `std::string`, the compiler will report an error.

Example 49.1 displays `Boost`.

Example 49.2. Specializing functions for a group of types with `boost::enable_if`

```
#include <boost/utility/enable_if.hpp>
#include <type_traits>
#include <iostream>

template <typename T>
void print(typename boost::enable_if<std::is_integral<T>, T>::type i)
{
  std::cout << "Integral: " << i << '\n';
}

template <typename T>
void print(typename boost::enable_if<std::is_floating_point<T>, T>::type f)
{
  std::cout << "Floating point: " << f << '\n';
}

int main()
{
  print<short>(1);
  print<long>(2);
  print<double>(3.14);
}
```

Example 49.2 uses `boost::enable_if` to specialize a function for a group of types. The function is called `print()` and expects one parameter. It can be overloaded, although overloading requires you to use a concrete type. To do the same for a group of types like `short`, `int` or `long`, you can define an appropriate condition using `boost::enable_if`. Example 49.2 uses `std::is_integral` to do so. The second `print()` function is overloaded with `std::is_floating_point` for all floating point numbers.

Exercise

Make `print_has_post_increment()` write to standard output whether a type supports the post-increment operator. For example, for `int` the program should output “int has a post increment operator”:

```
#include <string>

template <class T>
void print_has_post_increment()
{
    // TODO: Implement this function.
}

int main()
{
    print_has_post_increment<int>();
    print_has_post_increment<long>();
    print_has_post_increment<std::string>();
}
```