Chapter 43. Boost.Lambda

Before C++11, you needed to use a library like Boost.Lambda to take advantage of lambda functions. Since C++11, this library can be regarded as deprecated because lambda functions are now part of the programming language. If you work in a development environment that doesn't support C++11, you should consider Boost.Phoenix before you turn to Boost.Lambda. Boost.Phoenix is a newer library and probably the better choice if you need to use lambda functions without C++11.

The purpose of lambda functions is to make code more compact and easier to understand (see Example 43.1).

Example 43.1. `std::for_each()` with a lambda function

```
#include <boost/lambda/lambda.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(),
    std::cout << boost::lambda::_1 << "\n");
}
```

Boost.Lambda provides several helpers to create nameless functions. Code is written where it should be executed, without needing to be wrapped in a function and without having to call a function explicitly. In Example 43.1, `std::cout << boost::lambda::_1 << "\n"` is a lambda function that expects one parameter, which it writes, followed by a new line, to standard output.

`boost::lambda::_1` is a placeholder that creates a lambda function that expects one parameter. The number in the placeholder determines the number of expected parameters, so **`boost::lambda::_2`** expects two parameters and **`boost::lambda::_3`** expects three parameters. Boost.Lambda only provides these three placeholders. The lambda function in Example 43.1 uses **`boost::lambda::_1`** because `std::for_each()` expects a unary function.

Include `boost/lambda/lambda.hpp` to use placeholders.

Please note that `\n`, instead of `std::endl`, is used in Example 43.1 to output a new line. If you use `std::endl`, the example won't compile because the type of the lambda function `std::cout << boost::lambda::_1` differs from what the unary function template `std::endl()` expects. Thus, you can't use `std::endl`.

Example 43.2. A lambda function with `boost::lambda::if_then()`

```
#include <boost/lambda/lambda.hpp>
#include <boost/lambda/if.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
```

```
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(),
    boost::lambda::if_then(boost::lambda::_1 > 1,
    std::cout << boost::lambda::_1 << "\n"));
}
```

The header file `boost/lambda/if.hpp` defines constructs you can use to create `if` control structures in a lambda function. The simplest construct is the function template `boost::lambda::if_then()`, which expects two parameters: the first parameter is a condition. If the condition is true, the second parameter is executed. Both parameters can be lambda functions, as in Example 43.2.

In addition to `boost::lambda::if_then()`, Boost.Lambda provides the function templates `boost::lambda::if_then_else()` and `boost::lambda::if_then_else_return()`, both of which expect three parameters. Function templates are also provided `for` loops and cast operators and to throw exceptions in lambda functions. The many function templates defined by Boost.Lambda make it possible to define lambda functions that are in no way inferior to normal C++ functions.