Chapter 53. Boost.Parameter

Boost.Parameter makes it possible to pass parameters as key/value pairs. In addition to supporting function parameters, the library also supports template parameters. Boost.Parameter is especially useful if you are using long parameter lists, and the order and meaning of parameters is difficult to remember. Key/value pairs make it possible to pass parameters in any order. Because every value is passed with a key, the meaning of the various values is also clearer.

Example 53.1. Function parameters as key/value pairs

```
#include <boost/parameter.hpp>
#include <string>
#include <iostream>
#include <ios>

BOOST_PARAMETER_NAME(a)
BOOST_PARAMETER_NAME(b)
BOOST_PARAMETER_NAME(c)
BOOST_PARAMETER_NAME(d)
BOOST_PARAMETER_NAME(e)

BOOST_PARAMETER_FUNCTION(
  (void),
  complicated,
  tag,
  (required
    (a, (int))
    (b, (char))
    (c, (double))
    (d, (std::string))
    (e, *)
  )
)
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << a << '\n';
  std::cout << b << '\n';
  std::cout << c << '\n';
  std::cout << d << '\n';
  std::cout << e << '\n';
}

int main()
{
  complicated(_c = 3.14, _a = 1, _d = "Boost", _b = 'B', _e = true);
}
```

Example 53.1 defines a function `complicated()`, which expects five parameters. The parameters may be passed in any order. Boost.Parameter provides the macro `BOOST_PARAMETER_FUNCTION` to define such a function.

Before `BOOST_PARAMETER_FUNCTION` can be used, the parameters for the key/value pairs must be defined. This is done with the macro `BOOST_PARAMETER_NAME`, which is just passed a parameter name. The example uses `BOOST_PARAMETER_NAME` five times to define the parameter names **a**, **b**, **c**, **d**, and **e**.

Please note that the parameter names are automatically defined in the namespace `tag`. This should avoid clashes with identically named definitions in a program.

After the parameter names have been defined, `BOOST_PARAMETER_FUNCTION` is used to define the function `complicated()`. The first parameter passed to `BOOST_PARAMETER_FUNCTION` is the type of the return value. This is `void` in the example. Please note that the type must be wrapped in parentheses – the first parameter is `(void)`.

The second parameter is the name of the function being defined. The third parameter is the namespace containing the parameter names. In the fourth parameter, the parameter names are accessed to further specify them.

In Example 53.1 the fourth parameter starts with `required`, which is a keyword that makes the parameters that follow mandatory. `required` is followed by one or more pairs consisting of a parameter name and a type. It is important to wrap the type in parentheses.

Various types are used for the parameters **a**, **b**, **c**, and **d**. For example, **a** can be used to pass an `int` value to `complicated()`. No type is given for **e**. Instead, an asterisk is used, which means that the value passed may have any type. **e** is a template parameter.

After the various parameters have been passed to `BOOST_PARAMETER_FUNCTION`, the function body is defined. This is done, as usual, between a pair of curly brackets. Parameters can be accessed in the function body. They can be used like variables, with the types assigned within `BOOST_PARAMETER_FUNCTION`. Example 53.1 writes the parameters to standard output.

`complicated()` is called from `main()`. The parameters are passed to `complicated()` in an arbitrary order. Parameter names start with an underscore. Boost.Parameter uses the underscore to avoid name clashes with other variables.

Note

To pass function parameters as key/value pairs in C++, you can also use the named parameter idiom, which doesn't require a library like Boost.Parameter.

Example 53.2. Optional function parameters

```
#include <boost/parameter.hpp>
#include <string>
#include <iostream>
#include <ios>

BOOST_PARAMETER_NAME(a)
BOOST_PARAMETER_NAME(b)
BOOST_PARAMETER_NAME(c)
BOOST_PARAMETER_NAME(d)
BOOST_PARAMETER_NAME(e)

BOOST_PARAMETER_FUNCTION(
  (void),
  complicated,
  tag,
  (required
    (a, (int))
    (b, (char)))
  (optional
    (c, (double), 3.14)
    (d, (std::string), "Boost")
    (e, *, true))
)
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << a << '\n';
  std::cout << b << '\n';
  std::cout << c << '\n';
  std::cout << d << '\n';
  std::cout << e << '\n';
}

int main()
{
  complicated(_b = 'B', _a = 1);
}
```

`BOOST_PARAMETER_FUNCTION` also supports defining optional parameters.

In Example 53.2 the parameters **c**, **d**, and **e** are optional. These parameters are defined in `BOOST_PARAMETER_FUNCTION` using the `optional` keyword.

Optional parameters are defined like required parameters: a parameter name is given followed by a type. As usual, the type is wrapped in parentheses. However, optional parameters need to have a default value.

With the call to `complicated()`, only the parameters **a** and **b** are passed. These are the only required parameters. As the parameters **c**, **d**, and **e** aren’t used, they are set to default values.

Boost.Parameter provides macros in addition to `BOOST_PARAMETER_FUNCTION`. For example, you can use `BOOST_PARAMETER_MEMBER_FUNCTION` to define a member function, and `BOOST_PARAMETER_CONST_MEMBER_FUNCTION` to define a constant member function.

You can define functions with Boost.Parameter that try to assign values to parameters automatically. In that case, you don’t need to pass key/value pairs – it is sufficient to pass values only. If the types of all values are different, Boost.Parameter can detect which value belongs to which parameter. This might require you to have a deeper knowledge of template meta programming.

Example 53.3. Template parameters as key/value pairs

```
#include <boost/parameter.hpp>
#include <boost/mpl/placeholders.hpp>
#include <type_traits>
#include <typeinfo>
#include <iostream>

BOOST_PARAMETER_TEMPLATE_KEYWORD(integral_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(floating_point_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(any_type)

using namespace boost::parameter;
using boost::mpl::placeholders::_;

typedef parameters<
  required<tag::integral_type, std::is_integral<_>>,
  required<tag::floating_point_type, std::is_floating_point<_>>,
  required<tag::any_type, std::is_object<_>>
```

```
> complicated_signature;

template <class A, class B, class C>
class complicated
{
public:
  typedef typename complicated_signature::bind<A, B, C>::type args;
  typedef typename value_type<args, tag::integral_type>::type integral_type;
  typedef typename value_type<args, tag::floating_point_type>::type
    floating_point_type;
  typedef typename value_type<args, tag::any_type>::type any_type;
};

int main()
{
  typedef complicated<floating_point_type<double>, integral_type<int>,
    any_type<bool>> c;
  std::cout << typeid(c::integral_type).name() << '\n';
  std::cout << typeid(c::floating_point_type).name() << '\n';
  std::cout << typeid(c::any_type).name() << '\n';
}
```

Example 53.3 uses Boost.Parameter to pass template parameters as key/value pairs. As with functions, it is possible to pass the template parameters in any order.

The example defines a class `complicated`, which expects three template parameters. Because the order of the parameters doesn’t matter, they are called `A`, `B`, and `C`. `A`, `B`, and `C` aren’t the names of the parameters that will be used when the class template is accessed. As with functions, the parameter names are defined using a macro. For template parameters, `BOOST_PARAMETER_TEMPLATE_KEYWORD` is used. Example 53.3 defines three parameter names `integral_type`, `floating_point_type`, and `any_type`.

After the parameter names have been defined, you must specify the types that may be passed. For example, the parameter `integral_type` can be used to pass types such as `int` or `long`, but not a type like `std::string`. `boost::parameter::parameters` is used to create a signature that refers to the parameter names and defines which types may be passed with each of them.

`boost::parameter::parameters` is a tuple that describes parameters. Required parameters are marked with `boost::parameter::required`.

`boost::parameter::required` requires two parameters. The first is the name of the parameter defined with `BOOST_PARAMETER_TEMPLATE_KEYWORD`. The second identifies the type the parameter may be set to. For example, `integral_type` may be set to an integral type. This requirement is expressed with `std::is_integral<_>`. `std::is_integral<_>` is a lambda function based on Boost.MPL. `boost::mpl::placeholders::_` is a placeholder provided by this library. If the type to which `integral_type` is set is passed to `std::is_integral` instead of `boost::mpl::placeholders::_`, and the result is true, a valid type is used. The requirements for the other parameters `floating_point_type` and `any_type` are defined similarly.

After the signature has been created and defined as `complicated_signature`, it is used by the class `complicated`. First, the signature is bound with `complicated_signature::bind` to the template parameters `A`, `B`, and `C`. The new type, `args`, represents the connection between the template parameters passed and the requirements that must be met by the template parameters. Next, `args` is accessed to get the parameter values. This is done with `boost::parameter::value_type`. `boost::parameter::value_type` expects `args` and a parameter to be passed. The parameter determines the type created. In Example 53.3, the type definition `integral_type` in the class `complicated` is used to get the type that was passed with the parameter `integral_type` to `complicated`.

`main()` accesses `complicated` to instantiate the class. The parameter `integral_type` is set to `int`, `floating_point_type` to `double`, and `any_type` to `bool`. The order of the parameters passed doesn't matter. The type definitions `integral_type`, `floating_point_type`, and `any_type` are then accessed by `typeid` to get their underlying types. Compiled with Visual C++ 2013, the example writes `int`, `double` and `bool` to standard output.

Example 53.4. Optional template parameters

```cpp
#include <boost/parameter.hpp>
#include <boost/mpl/placeholders.hpp>
#include <type_traits>
#include <typeinfo>
#include <iostream>

BOOST_PARAMETER_TEMPLATE_KEYWORD(integral_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(floating_point_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(any_type)

using namespace boost::parameter;
using boost::mpl::placeholders::_;

typedef parameters<
  required<tag::integral_type, std::is_integral<_>>,
  optional<tag::floating_point_type, std::is_floating_point<_>>,
  optional<tag::any_type, std::is_object<_>>
> complicated_signature;

template <class A, class B = void_, class C = void_>
class complicated
{
public:
  typedef typename complicated_signature::bind<A, B, C>::type args;
  typedef typename value_type<args, tag::integral_type>::type integral_type;
  typedef typename value_type<args, tag::floating_point_type, float>::type
    floating_point_type;
  typedef typename value_type<args, tag::any_type, bool>::type any_type;
};

int main()
{
  typedef complicated<floating_point_type<double>, integral_type<short>> c;
  std::cout << typeid(c::integral_type).name() << '\n';
  std::cout << typeid(c::floating_point_type).name() << '\n';
  std::cout << typeid(c::any_type).name() << '\n';
}
```

Example 53.4 introduces optional template parameters. The signature uses `boost::parameter::optional` for the optional template parameters. The optional template parameters from `complicated` are set to `boost::parameter::void_`, and `boost::parameter::value_type` is given a default value. This default value is the type an optional parameter will be set to if the type isn't otherwise set.

complicated is instantiated in main(). This time only the parameters integral_type and floating_point_type are used. any_type is not used. Compiled with Visual C++ 2013, the example writes short for integral_type, double for floating_point_type, and bool for any_type to standard output.

Boost.Parameter can automatically detect template parameters. You can create signatures that allow types to be automatically assigned to parameters. As with function parameters, deeper knowledge in template meta programming is required to do this.