Chapter 48. Boost.TypeTraits

Types have different properties that generic programming takes advantage of. The Boost.TypeTraits library provides the tools needed to determine a type’s properties and change them.

Since C++11, some functions provided by Boost.TypeTraits can be found in the standard library. You can access those functions through the header file `type_traits`. However, Boost.TypeTraits provides additional functions.

Example 48.1. Determining type categories

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

using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_integral<int>::value << '\n';
  std::cout << is_floating_point<int>::value << '\n';
  std::cout << is_arithmetic<int>::value << '\n';
  std::cout << is_reference<int>::value << '\n';
}
```

Example 48.1 calls several functions to determine type categories. `boost::is_integral` checks whether a type is integral – whether it can store integers. `boost::is_floating_point` checks whether a type stores floating point numbers. `boost::is_arithmetic` checks whether a type supports arithmetic operators. And `boost::is_reference` can be used to determine whether a type is a reference.

`boost::is_integral` and `boost::is_floating_point` are mutually exclusive. A type either stores an integer or a floating point number. However, `boost::is_arithmetic` and `boost::is_reference` can apply to multiple categories. For example, both integer and floating point types support arithmetic operations.

All functions from Boost.TypeTraits provide a result in **`value`** that is either `true` or `false`. Example 48.1 outputs `true` for `is_integral<int>` and `is_arithmetic<int>` and outputs `false` for `is_floating_point<int>` and `is_reference<int>`. Because all of these functions are templates, nothing is processed at run time. The example behaves at run time as though the values `true` and `false` were directly used in the code.

In Example 48.1, the result of the various functions is a value of type `bool`, which can be written directly to standard output. If the result is to be processed by a function template, it should be forwarded as a type, not as a `bool` value.

Example 48.2. `boost::true_type` and `boost::false_type`

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

```
using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_same<is_integral<int>::type, true_type>::value << '\n';
  std::cout << is_same<is_floating_point<int>::type, false_type>::value <<
    '\n';
  std::cout << is_same<is_arithmetic<int>::type, true_type>::value << '\n';
  std::cout << is_same<is_reference<int>::type, false_type>::value << '\n';
}
```

Besides **value**, functions from Boost.TypeTraits also provide the result in `type`. While **value** is a `bool` value, `type` is a type. Just like **value**, which can only be set to `true` or `false`, `type` can only be set to one of two types: `boost::true_type` or `boost::false_type`. `type` lets you pass the result of a function as a type to another function.

Example 48.2 uses another function from Boost.TypeTraits called `boost::is_same`. This function expects two types as parameters and checks whether they are the same. To pass the results of `boost::is_integral`, `boost::is_floating_point`, `boost::is_arithmetic`, and `boost::is_reference` to `boost::is_same`, `type` must be accessed. `type` is then compared with `boost::true_type` or `boost::false_type`. The results from `boost::is_same` are then read through **value** again. Because this is a `bool` value, it can be written to standard output.

Example 48.3. Checking type properties with Boost.TypeTraits

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

using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << has_plus<int>::value << '\n';
  std::cout << has_pre_increment<int>::value << '\n';
  std::cout << has_trivial_copy<int>::value << '\n';
  std::cout << has_virtual_destructor<int>::value << '\n';
}
```

Example 48.3 introduces functions that check properties of types. `boost::has_plus` checks whether a type supports the operator `operator+` and whether two objects of the same type can be concatenated. `boost::has_pre_increment` checks whether a type supports the pre-increment operator `operator++`. `boost::has_trivial_copy` checks whether a type has a trivial copy constructor. And `boost::has_virtual_destructor` checks whether a type has a virtual destructor.

Example 48.3 displays `true` three times and `false` once.

Example 48.4. Changing type properties with Boost.TypeTraits

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

using namespace boost;

int main()
```

```
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_const<add_const<int>::type>::value << '\n';
  std::cout << is_same<remove_pointer<int*>::type, int>::value << '\n';
  std::cout << is_same<make_unsigned<int>::type, unsigned int>::value <<
    '\n';
  std::cout << is_same<add_rvalue_reference<int>::type, int&&>::value <<
    '\n';
}
```

Example 48.4 illustrates how type properties can be changed. `boost::add_const` adds `const` to a type. If the type is already constant, nothing changes. The code compiles without problems, and the type remains constant.

`boost::remove_pointer` removes the asterisk from a pointer type and returns the type the pointer refers to. `boost::make_unsigned` turns a type with a sign into a type without a sign. And `boost::add_rvalue_reference` transforms a type into a rvalue reference.

Example 48.4 writes `true` four times to standard output.