Chapter 69. Boost.Utility

The library Boost.Utility is a conglomeration of miscellaneous, useful classes and functions that are too small to justify being maintained in stand-alone libraries. While the utilities are small and can be learned quickly, they are completely unrelated. Unlike the examples in other chapters, the code samples here do not build on each other, since they are independent utilities.

While most utilities are defined in `boost/utility.hpp`, some have their own header files. The following examples include the appropriate header file for the utility being introduced.

Example 69.1. Using `boost::checked_delete()`

```
#include <boost/checked_delete.hpp>
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

struct animal : public boost::intrusive::list_base_hook<>
{
  std::string name_;
  int legs_;

  animal(std::string name, int legs) : name_{std::move(name)},
    legs_{legs} {}
};

int main()
{
  animal *a = new animal{"cat", 4};

  typedef boost::intrusive::list<animal> animal_list;
  animal_list al;

  al.push_back(*a);

  al.pop_back_and_dispose(boost::checked_delete<animal>);
  std::cout << al.size() << '\n';
}
```

Example 69.1 passes the function `boost::checked_delete()` as a parameter to the member function `pop_back_and_dispose()`, which is provided by the class `boost::intrusive::list` from Boost.Intrusive. `boost::intrusive::list` and `pop_back_and_dispose()` are introduced in Chapter 18, while `boost::checked_delete()` is provided by Boost.Utility and defined in `boost/checked_delete.hpp`.

`boost::checked_delete()` expects as its sole parameter a pointer to the object that will be deleted by `delete`. Because `pop_back_and_dispose()` expects a function that takes a pointer to destroy the corresponding object, it makes sense to pass in `boost::checked_delete()` – that way, you don't need to define a similar function.

Unlike `delete`, `boost::checked_delete()` ensures that the type of the object to be destroyed is complete. `delete` will accept a pointer to an object with an incomplete type. While this concerns a detail of the C++ standard that you can usually ignore, you should note that

`boost::checked_delete()` is not completely identical to a call to `delete` because it puts higher demands on its parameter.

Boost.Utility also provides `boost::checked_array_delete()`, which can be used to destroy arrays. It calls `delete[]` rather than `delete`.

Additionally, two classes, `boost::checked_deleter` and `boost::checked_array_deleter`, are available to create function objects that behave like `boost::checked_delete()` and `boost::checked_array_delete()`, respectively.

Example 69.2. Using `BOOST_CURRENT_FUNCTION`

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

int main()
{
  const char *funcname = BOOST_CURRENT_FUNCTION;
  std::cout << funcname << '\n';
}
```

Example 69.2 uses the macro `BOOST_CURRENT_FUNCTION`, defined in `boost/current_function.hpp`, to return the name of the surrounding function as a string.

`BOOST_CURRENT_FUNCTION` provides a platform-independent way to retrieve the name of a function. Starting with C++11, you can do the same thing with the standardized macro `__func__`. Before C++11, compilers like Visual C++ and GCC supported the macro `__FUNCTION__` as an extension. `BOOST_CURRENT_FUNCTION` uses whatever macro is supported by the compiler.

If compiled with Visual C++ 2013, Example 69.2 displays `int __cdecl main(void)`.

Example 69.3. Using `boost::prior()` and `boost::next()`

```
#include <boost/next_prior.hpp>
#include <array>
#include <algorithm>
#include <iostream>

int main()
{
  std::array<char, 4> a{{'a', 'c', 'b', 'd'}};

  auto it = std::find(a.begin(), a.end(), 'b');
  auto prior = boost::prior(it, 2);
  auto next = boost::next(it);

  std::cout << *prior << '\n';
  std::cout << *it << '\n';
  std::cout << *next << '\n';
}
```

Boost.Utility provides two functions, `boost::prior()` and `boost::next()`, that return an iterator relative to another iterator. In Example 69.3, **`it`** points to “b” in the array, **`prior`** points to “a”, and **`next`** to “d”.

Unlike `std::advance()`, `boost::prior()` and `boost::next()` return a new iterator and do not modify the iterator that was passed in.

In addition to the iterator, both functions accept a second parameter that indicates the number of steps to move forward or backward. In Example 69.3, the iterator is moved two steps backward in the call to `boost::prior()` and one step forward in the call to `boost::next()`.

The number of steps is always a positive number, even for `boost::prior()`, which moves backwards.

To use `boost::prior()` and `boost::next()`, include the header file `boost/next_prior.hpp`.

Both functions were added to the standard library in C++11, where they are called `std::prev()` and `std::next()`. They are defined in the header file `iterator`.

Example 69.4. Using `boost::noncopyable`

```
#include <boost/noncopyable.hpp>
#include <string>
#include <utility>
#include <iostream>

struct animal : boost::noncopyable
{
  std::string name;
  int legs;

  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

void print(const animal &a)
{
  std::cout << a.name << '\n';
  std::cout << a.legs << '\n';
}

int main()
{
  animal a{"cat", 4};
  print(a);
}
```

Boost.Utility provides the class `boost::noncopyable`, which is defined in `boost/noncopyable.hpp`. This class makes it impossible to copy (and move) objects.

The same effect can be achieved by defining the copy constructor and assignment operator as private member functions or – since C++11 – by removing the copy constructor and assignment operator with `delete`. However, deriving from `boost::noncopyable` explicitly states the intention that objects of a class should be non-copyable.

Note

Some developers prefer `boost::noncopyable` while others prefer to remove member functions explicitly with `delete`. You will find arguments for both approaches at Stack Overflow, among other places.

Example 69.4 can be compiled and executed. However, if the signature of the `print()` function is modified to take an object of type `animal` by value rather than by reference, the resulting code will no longer compile.

Example 69.5. Using `boost::addressof()`

```
#include <boost/utility/addressof.hpp>
#include <string>
#include <iostream>

struct animal
{
  std::string name;
  int legs;

  int operator&() const { return legs; }
};

int main()
{
  animal a{"cat", 4};
  std::cout << &a << '\n';
  std::cout << boost::addressof(a) << '\n';
}
```

To retrieve the address of a particular object, even if `operator&` has been overloaded, Boost.Utility provides the function `boost::addressof()`, which is defined in `boost/utility/addressof.hpp` (see Example 69.5). With C++11, this function became part of the standard library and is available as `std::addressof()` in the header file `memory`.

Example 69.6. Using `BOOST_BINARY`

```
#include <boost/utility/binary.hpp>
#include <iostream>

int main()
{
  int i = BOOST_BINARY(1001 0001);
  std::cout << i << '\n';

  short s = BOOST_BINARY(1000 0000 0000 0000);
  std::cout << s << '\n';
}
```

The macro `BOOST_BINARY` lets you create numbers in binary form. Standard C++ only supports hexadecimal and octal forms, using the prefixes `0x` and `0`. C++11 introduced user-defined literals, which allows you to define custom suffixes, but there still is no standard way of using numbers in binary form in C++11.

Example 69.6 displays `145` and `-32768`. The bit sequence stored in **s** represents a negative number because the 16-bit type `short` uses the 16th bit – the most significant bit in `short` – as the sign bit.

`BOOST_BINARY` simply offers another option to write numbers. Because, in C++, the default type for numbers is `int`, `BOOST_BINARY` also uses `int`. To define a number of type `long`, use the macro `BOOST_BINARY_L`, which generates the equivalent of a number suffixed with the letter L.

Boost.Utility includes additional macros such as `BOOST_BINARY_U`, which initializes a variable without a sign bit. All of these macros are defined in the header file `boost/utility/binary.hpp`.

Example 69.7. Using `boost::string_ref`

```
#include <boost/utility/string_ref.hpp>
#include <iostream>

boost::string_ref start_at_boost(boost::string_ref s)
{
  auto idx = s.find("Boost");
  return (idx != boost::string_ref::npos) ? s.substr(idx) : "";
}

int main()
{
  boost::string_ref s = "The Boost C++ Libraries";
  std::cout << start_at_boost(s) << '\n';
}
```

Example 69.7 introduces the class `boost::string_ref`, which is a reference to a string that only supports read access. To a certain extent, the reference is comparable with `const std::string&`. However, `const std::string&` requires the existence of an object of type `std::string`. `boost::string_ref` can also be used without `std::string`. The benefit of `boost::string_ref` is that, unlike `std::string`, it requires no memory to be allocated.

Example 69.7 looks for the word “Boost” in a string. If found, a string starting with that word is displayed. If the word “Boost” isn’t found, an empty string is displayed. The type of the string **s** in `main()` isn’t `std::string`, it’s `boost::string_ref`. Thus no memory is allocated with `new` and no copy is created. **s** points to the literal string “The Boost C++ Libraries” directly.

The type of the return value of `start_at_boost()` is `boost::string_ref`, not `std::string`. The function doesn’t return a new string, it returns a reference. This reference is to either a substring of the parameter or an empty string. `start_at_boost()` requires that the original string remains valid as long as references of type `boost::string_ref` are in use. If this is guaranteed, as in Example 69.7, memory allocations can be avoided.

Additional utilities are also available, but they are beyond the scope of this book because they are mostly used by the developers of Boost libraries or for template meta programming. The documentation of Boost.Utility provides a fairly comprehensive overview of these additional utilities and can serve as a starting point if you are interested.