Chapter 41. Boost.Bind

Boost.Bind is a library that simplifies and generalizes capabilities that originally required `std::bind1st()` and `std::bind2nd()`. These two functions were added to the standard library with C++98 and made it possible to connect functions even if their signatures aren’t compatible.

Boost.Bind was added to the standard library with C++11. If your development environment supports C++11, you will find the function `std::bind()` in the header file `functional`. Depending on the use case, it may be better to use lambda functions or Boost.Phoenix than `std::bind()` or Boost.Bind.

Example 41.1. `std::for_each()` with a compatible function

```
#include <vector>
#include <algorithm>
#include <iostream>

void print(int i)
{
  std::cout << i << '\n';
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), print);
}
```

The third parameter of `std::for_each()` is a function or function object that expects a sole parameter. In Example 41.1, `std::for_each()` passes the numbers in the container **v** as sole parameters, one after another, to `print()`.

If you need to pass in a function whose signature doesn’t meet the requirements of an algorithm, it gets more difficult. For example, if you want `print()` to accept an output stream as an additional parameter, you can no longer use it as is with `std::for_each()`.

Example 41.2. `std::for_each()` with `std::bind1st()`

```
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

class print : public std::binary_function<std::ostream*, int, void>
{
public:
  void operator()(std::ostream *os, int i) const
  {
    *os << i << '\n';
  }
};

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), std::bind1st(print{}, &std::cout));
}
```

Like Example 41.1, Example 41.2 writes all numbers in **v** to standard output. However, this time, the output stream is passed to `print()` as a parameter. To do this, the function `print()` is defined as a function object derived from `std::binary_function`.

With Boost.Bind, you don’t need to transform `print()` from a function to a function object. Instead, you use the function template `boost::bind()`, which is defined in `boost/bind.hpp`.

Example 41.3. `std::for_each()` with `boost::bind()`

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

void print(std::ostream *os, int i)
{
  *os << i << '\n';
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), boost::bind(print, &std::cout, _1));
}
```

Example 41.3 uses `print()` as a function, not as a function object. Because `print()` expects two parameters, the function can’t be passed directly to `std::for_each()`. Instead, `boost::bind()` is passed to `std::for_each()` and `print()` is passed as the first parameter to `boost::bind()`.

Since `print()` expects two parameters, those two parameters must also be passed to `boost::bind()`. They are a pointer to **`std::cout`** and **_1**.

_1 is a placeholder. Boost.Bind defines placeholders from **_1** to **_9**. These placeholders tell `boost::bind()` to return a function object that expects as many parameters as the placeholder with the greatest number. If, as in Example 41.3, only the placeholder **_1** is used, `boost::bind()` returns an unary function object – a function object that expects a sole parameter. This is required in this case since `std::for_each()` passes only one parameter.

`std::for_each()` calls a unary function object. The value passed to the function object – a number from the container **v** – takes the position of the placeholder **_1**. `boost::bind()` takes the number and the pointer to **`std::cout`** and forwards them to `print()`.

Please note that `boost::bind()`, like `std::bind1st()` and `std::bind2nd()`, takes parameters by value. To prevent the calling program from trying to copy **`std::cout`**, `print()` expects a pointer to a stream. Boost.Ref provides a function which allows you to pass a parameter by reference.

Example 41.4 illustrates how to define a binary function object with `boost::bind()`. It uses the algorithm `std::sort()`, which expects a binary function as its third parameter.

Example 41.4. `std::sort()` with `boost::bind()`

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

bool compare(int i, int j)
{
  return i > j;
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::sort(v.begin(), v.end(), boost::bind(compare, _1, _2));
  for (int i : v)
    std::cout << i << '\n';
}
```

In Example 41.4, a binary function object is created because the placeholder **_2** is used. The algorithm `std::sort()` calls this binary function object with two values from the container **v** and evaluates the return value to sort the container. The function `compare()` is defined to sort **v** in descending order.

Since `compare()` is a binary function, it can be passed to `std::sort()` directly. However, it can still make sense to use `boost::bind()` because it lets you change the order of the parameters. For example, you can use `boost::bind()` if you want to sort the container in ascending order but don’t want to change `compare()`.

Example 41.5. `std::sort()` with `boost::bind()` and changed order of placeholders

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

bool compare(int i, int j)
{
  return i > j;
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::sort(v.begin(), v.end(), boost::bind(compare, _2, _1));
  for (int i : v)
    std::cout << i << '\n';
}
```

Example 41.5 sorts **v** in ascending order simply by swapping the placeholders: **_2** is passed first and **_1** second.