Chapter 39. Boost.Phoenix

Boost.Phoenix is the most important Boost library for functional programming. While libraries like Boost.Bind or Boost.Lambda provide some support for functional programming, Boost.Phoenix includes the features of these libraries and goes beyond them.

In functional programming, functions are objects and can be processed like objects. With Boost.Phoenix, it is possible for a function to return another function as a result. It is also possible to pass a function as a parameter to another function. Because functions are objects, it's possible to distinguish between instantiation and execution. Accessing a function isn't equal to executing it.

Boost.Phoenix supports functional programming with function objects: Functions are objects based on classes which overload the operator `operator()`. That way function objects behave like other objects in C++. For example, they can be copied and stored in a container. However, they also behave like functions because they can be called.

Functional programming isn't new in C++. You can pass a function as a parameter to another function without using Boost.Phoenix.

Example 39.1. Predicates as global function, lambda function, and Phoenix function

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

bool is_odd(int i) { return i % 2 == 1; }

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  std::cout << std::count_if(v.begin(), v.end(), is_odd) << '\n';

  auto lambda = [](int i){ return i % 2 == 1; };
  std::cout << std::count_if(v.begin(), v.end(), lambda) << '\n';

  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

Example 39.1 uses the algorithm `std::count_if()` to count odd numbers in vector **v**. `std::count_if()` is called three times, once with a predicate as a free-standing function, once with a lambda function, and once with a Phoenix function.

The Phoenix function differs from free-standing and lambda functions because it has no frame. While the other two functions have a function header with a signature, the Phoenix function seems to consist of a function body only.

The crucial component of the Phoenix function is **`boost::phoenix::placeholders::arg1`**. **`arg1`** is a global instance of a function object. You can use it like **`std::cout`**: These objects exist

once the respective header file is included.

arg1 is used to define an unary function. The expression `arg1 % 2 == 1` creates a new function that expects one parameter. The function isn't executed immediately but stored in **phoenix**. **phoenix** is passed to `std::count_if()` which calls the predicate for every number in **v**.

arg1 is a placeholder for the value passed when the Phoenix function is called. Since only **arg1** is used here, a unary function is created. Boost.Phoenix provides additional placeholders such as **boost::phoenix::placeholders::arg2** and **boost::phoenix::placeholders::arg3**. A Phoenix function always expects as many parameters as the placeholder with the greatest number.

Example 39.1 writes `3` three times to standard output.

Example 39.2. Phoenix function versus lambda function

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

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  auto lambda = [](int i){ return i % 2 == 1; };
  std::cout << std::count_if(v.begin(), v.end(), lambda) << '\n';

  std::vector<long> v2;
  v2.insert(v2.begin(), v.begin(), v.end());

  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
  std::cout << std::count_if(v2.begin(), v2.end(), phoenix) << '\n';
}
```

Example 39.2 highlights a crucial difference between Phoenix and lambda functions. In addition to requiring no function header with a parameter list, Phoenix function parameters have no types. The lambda function **lambda** expects a parameter of type `int`. The Phoenix function **phoenix** will accept any type that the modulo operator can handle.

Think of Phoenix functions as function templates. Like function templates, Phoenix functions can accept any type. This makes it possible in Example 39.2 to use **phoenix** as a predicate for the containers **v** and **v2** even though they store numbers of different types. If you try to use the predicate **lambda** with **v2**, you get a compiler error.

Example 39.3. Phoenix functions as deferred C++ code

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

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};
```

```
  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 > 2 && arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

Example 39.3 uses a Phoenix function as a predicate with `std::count_if()` to count odd numbers greater than 2. The Phoenix function accesses **`arg1`** twice: Once to test if the placeholder is greater than 2 and once to test whether it's an odd number. The conditions are linked with `&&`.

You can think of Phoenix functions as C++ code that isn't executed immediately. The Phoenix function in Example 39.3 looks like a condition that uses multiple logical and arithmetic operators. However, the condition isn't executed immediately. It is only executed when it is accessed from within `std::count_if()`. The access in `std::count_if()` is a normal function call.

Example 39.3 writes `2` to standard output.

Example 39.4. Explicit Phoenix types

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

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 > val(2) && arg1 % val(2) == val(1);
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

Example 39.4 uses explicit types for all operands in the Phoenix function. Strictly speaking, you don't see types, just the helper function `boost::phoenix::val()`. This function returns a function object initialized with the values passed to `boost::phoenix::val()`. The actual type of the function object doesn't matter. What is important is that Boost.Phoenix overloads operators like `>`, `&&`, `%` and `==` for different types. Thus, conditions aren't checked immediately. Instead, function objects are combined to create more powerful function objects. Depending on the operands they may be automatically used as function objects. Otherwise you can call helper functions like `val()`.

Example 39.5. **`boost::phoenix::placeholders::arg1`** and `boost::phoenix::val()`

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

int main()
{
  using namespace boost::phoenix::placeholders;
  std::cout << arg1(1, 2, 3, 4, 5) << '\n';

  auto v = boost::phoenix::val(2);
  std::cout << v() << '\n';
}
```

Example 39.5 illustrates how **arg1** and `val()` work. **arg1** is an instance of a function object. It can be used directly and called like a function. You can pass as many parameters as you like – **arg1** returns the first one.

`val()` is a function to create an instance of a function object. The function object is initialized with the value passed as a parameter. If the instance is accessed like a function, the value is returned.

Example 39.5 writes 1 and 2 to standard output.

Example 39.6. Creating your own Phoenix functions

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

struct is_odd_impl
{
    typedef bool result_type;

    template <typename T>
    bool operator()(T t) const { return t % 2 == 1; }
};

boost::phoenix::function<is_odd_impl> is_odd;

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), is_odd(arg1)) << '\n';
}
```

Example 39.6 explains how you can create your own Phoenix function. You pass a function object to the template `boost::phoenix::function`. The example passes the class `is_odd_impl`. This class overloads the operator `operator()`: when an odd number is passed in, the operator returns `true`. Otherwise, the operator returns `false`.

Please note that you must define the type `result_type`. Boost.Phoenix uses it to detect the type of the return value of the operator `operator()`.

`is_odd()` is a function you can use like `val()`. Both functions return a function object. When called, parameters are forwarded to the operator `operator()`. For Example 39.6, this means that `std::count_if()` still counts odd numbers.

Example 39.7. Transforming free-standing functions into Phoenix functions

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

bool is_odd_function(int i) { return i % 2 == 1; }

BOOST_PHOENIX_ADAPT_FUNCTION(bool, is_odd, is_odd_function, 1)
```

```
int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), is_odd(arg1)) << '\n';
}
```

If you want to transform a free-standing function into a Phoenix function, you can proceed as in Example 39.7. You don't necessarily have to define a function object as in the previous example.

You use the macro `BOOST_PHOENIX_ADAPT_FUNCTION` to turn a free-standing function into a Phoenix function. Pass the type of the return value, the name of the Phoenix function to define, the name of the free-standing function, and the number of parameters to the macro.

Example 39.8. Phoenix functions with `boost::phoenix::bind()`

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

bool is_odd(int i) { return i % 2 == 1; }

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), bind(is_odd, arg1)) << '\n';
}
```

To use a free-standing function as a Phoenix function, you can also use `boost::phoenix::bind()` as in Example 39.8. `boost::phoenix::bind()` works like `std::bind()`. The name of the free-standing function is passed as the first parameter. All further parameters are forwarded to the free-standing function.

Tip

Avoid `boost::phoenix::bind()`. Create your own Phoenix functions. This leads to more readable code. Especially with complex expressions it's not helpful having to deal with the additional details of `boost::phoenix::bind()`.

Example 39.9. Arbitrarily complex Phoenix functions

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

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  int count = 0;
```

```
  std::for_each(v.begin(), v.end(), if_(arg1 > 2 && arg1 % 2 == 1)
    [
      ++ref(count)
    ]);
  std::cout << count << '\n';
}
```

Boost.Phoenix provides some function objects that simulate C++ keywords. For example, you can use the function `boost::phoenix::if_()` (see Example 39.9) to create a function object that acts like `if` and tests a condition. If the condition is true, the code passed to the function object with `operator[]` will be executed. Of course, that code also has to be based on function objects. That way, you can create complex Phoenix functions.

Example 39.9 increments **`count`** for every odd number greater than 2. To use the increment operator on **`count`**, **`count`** is wrapped in a function object using `boost::phoenix::ref()`. In contrast to `boost::phoenix::val()`, no value is copied into the function object. The function object returned by `boost::phoenix::ref()` stores a reference – here a reference to **`count`**.

Tip

Don’t use Boost.Phoenix to create complex functions. It is better to use lambda functions from C++11. While Boost.Phoenix comes close to C++ syntax, using keywords like `if_` or code blocks between square brackets doesn’t necessarily improve readability.