Chapter 51. Boost.Coroutine

With Boost.Coroutine it is possible to use *coroutines* in C++. Coroutines are a feature of other programming languages, which often use the keyword `yield` for coroutines. In these programming languages, `yield` can be used like `return`. However, when `yield` is used, the function remembers the location, and if the function is called again, execution continues from that location.

C++ doesn't define a keyword `yield`. However, with Boost.Coroutine it is possible to return from functions and continue later from the same location. The Boost.Asio library also uses Boost.Coroutine and benefits from coroutines.

There are two versions of Boost.Coroutine. This chapter introduces the second version, which is the current version. This version has been available since Boost 1.55.0 and replaces the first one.

Example 51.1. Using coroutines

```
#include <boost/coroutine/all.hpp>
#include <iostream>

using namespace boost::coroutines;

void cooperative(coroutine<void>::push_type &sink)
{
  std::cout << "Hello";
  sink();
  std::cout << "world";
}

int main()
{
  coroutine<void>::pull_type source{cooperative};
  std::cout << ", ";
  source();
  std::cout << "!\n";
}
```

Example 51.1 defines a function, `cooperative()`, which is called from `main()` as a coroutine. `cooperative()` returns to `main()` early and is called a second time. On the second call, it continues from where it left off.

To use `cooperative()` as a coroutine, the types `pull_type` and `push_type` are used. These types are provided by `boost::coroutines::coroutine`, which is a template that is instantiated with `void` in Example 51.1.

To use coroutines, you need `pull_type` and `push_type`. One of these types will be used to create an object that will be initialized with the function you want to use as a coroutine. The other type will be the first parameter of the coroutine function.

Example 51.1 creates an object named **`source`** of type `pull_type` in `main()`. `cooperative()` is passed to the constructor. `push_type` is used as the sole parameter in the signature of `cooperative()`.

When **source** is created, the function `cooperative()`, which is passed to the constructor, is immediately called as a coroutine. This happens because **source** is based on `pull_type`. If **source** was based on `push_type`, the constructor wouldn't call `cooperative()` as a coroutine.

`cooperative()` writes `Hello` to standard output. Afterwards, the function accesses **sink** as if it were a function. This is possible because `push_type` overloads `operator()`. While **source** in `main()` represents the coroutine `cooperative()`, **sink** in `cooperative()` represents the function `main()`. Calling **sink** makes `cooperative()` return, and `main()` continues from where `cooperative()` was called and writes a comma to standard output.

Then, `main()` calls **source** as if it were a function. Again, this is possible because of the overloaded `operator()`. This time, `cooperative()` continues from the point where it left off and writes `world` to standard output. Because there is no other code in `cooperative()`, the coroutine ends. It returns to `main()`, which writes an exclamation mark to standard output.

The result is that Example 51.1 displays `Hello, world!`

You can think of coroutines as cooperative threads. To a certain extent, the functions `main()` and `cooperative()` run concurrently. Code is executed in turns in `main()` and `cooperative()`. Instructions inside each function are executed sequentially. Thanks to coroutines, a function doesn't need to return before another function can be executed.

Example 51.2. Returning a value from a coroutine

```
#include <boost/coroutine/all.hpp>
#include <functional>
#include <iostream>

using boost::coroutines::coroutine;

void cooperative(coroutine<int>::push_type &sink, int i)
{
  int j = i;
  sink(++j);
  sink(++j);
  std::cout << "end\n";
}

int main()
{
  using std::placeholders::_1;
  coroutine<int>::pull_type source{std::bind(cooperative, _1, 0)};
  std::cout << source.get() << '\n';
  source();
  std::cout << source.get() << '\n';
  source();
}
```

Example 51.2 is similar to the previous example. This time the template `boost::coroutines::coroutine` is instantiated with `int`. This makes it possible to return an `int` from the coroutine to the caller.

The direction the `int` value is passed depends on where `pull_type` and `push_type` are used. The example uses `pull_type` to instantiate an object in `main()`. `cooperative()` has access to

an object of type `push_type`. `push_type` sends a value, and `pull_type` receives a value; thus, the direction of the data transfer is set.

`cooperative()` calls **`sink`**, with a parameter of type `int`. This parameter is required because the coroutine was instantiated with the data type `int`. The value passed to **`sink`** is received from **`source`** in `main()` by using the member function `get()`, which is provided by `pull_type`.

Example 51.2 also illustrates how a function with multiple parameters can be used as a coroutine. `cooperative()` has an additional parameter of type `int`, which can't be passed directly to the constructor of `pull_type`. The example uses `std::bind()` to link the function with `pull_type`.

The example writes `1` and `2` followed by `end` to standard output.

Example 51.3. Passing two values to a coroutine

```
#include <boost/coroutine/all.hpp>
#include <tuple>
#include <string>
#include <iostream>

using boost::coroutines::coroutine;

void cooperative(coroutine<std::tuple<int, std::string>>::pull_type &source)
{
  auto args = source.get();
  std::cout << std::get<0>(args) << " " << std::get<1>(args) << '\n';
  source();
  args = source.get();
  std::cout << std::get<0>(args) << " " << std::get<1>(args) << '\n';
}

int main()
{
  coroutine<std::tuple<int, std::string>>::push_type sink{cooperative};
  sink(std::make_tuple(0, "aaa"));
  sink(std::make_tuple(1, "bbb"));
  std::cout << "end\n";
}
```

Example 51.3 uses `push_type` in `main()` and `pull_type` in `cooperative()`, which means data is transferred from the caller to the coroutine.

This example illustrates how multiple values can be passed. Boost.Coroutine doesn't support passing multiple values, so a tuple must be used. You need to pack multiple values into a tuple or another structure.

Example 51.3 displays `0 aaa`, `1 bbb`, and `end`.

Example 51.4. Coroutines and exceptions

```
#include <boost/coroutine/all.hpp>
#include <stdexcept>
#include <iostream>

using boost::coroutines::coroutine;

void cooperative(coroutine<void>::push_type &sink)
{
```

```
  sink();
  throw std::runtime_error("error");
}

int main()
{
  coroutine<void>::pull_type source{cooperative};
  try
  {
    source();
  }
  catch (const std::runtime_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

A coroutine returns immediately when an exception is thrown. The exception is transported to the caller of the coroutine where it can be caught. Thus, exceptions are no different than with regular function calls.

Example 51.4 shows how this works. This example will write the string `error` to standard output.