Chapter 3. Boost.ScopeExit

The library Boost.ScopeExit makes it possible to use RAII without resource-specific classes.

Example 3.1. Using `BOOST_SCOPE_EXIT`

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

int *foo()
{
  int *i = new int{10};
  BOOST_SCOPE_EXIT(&i)
  {
    delete i;
    i = 0;
  } BOOST_SCOPE_EXIT_END
  std::cout << *i << '\n';
  return i;
}

int main()
{
  int *j = foo();
  std::cout << j << '\n';
}
```

Boost.ScopeExit provides the macro `BOOST_SCOPE_EXIT`, which can be used to define something that looks like a local function but doesn't have a name. However, it does have a parameter list in parentheses and a block in braces.

The header file `boost/scoped_exit.hpp` must be included to use `BOOST_SCOPE_EXIT`.

The parameter list for the macro contains variables from the outer scope which should be accessible in the block. The variables are passed by copy. To pass a variable by reference, it must be prefixed with an ampersand, as in Example 3.1.

Code in the block can only access variables from the outer scope if the variables are in the parameter list.

`BOOST_SCOPE_EXIT` is used to define a block that will be executed when the scope the block is defined in ends. In Example 3.1 the block defined with `BOOST_SCOPE_EXIT` is executed just before `foo()` returns.

`BOOST_SCOPE_EXIT` can be used to benefit from RAII without having to use resource-specific classes. `foo()` uses `new` to create an `int` variable. In order to free the variable, a block that calls `delete` is defined with `BOOST_SCOPE_EXIT`. This block is guaranteed to be executed even if, for example, the function returns early because of an exception. In Example 3.1, `BOOST_SCOPE_EXIT` is as good as a smart pointer.

Please note that the variable **`i`** is set to 0 at the end of the block defined by `BOOST_SCOPE_EXIT`. **`i`** is then returned by `foo()` and written to the standard output stream in `main()`. However, the example doesn't display 0. **`j`** is set to a random value – namely the address where the `int` variable was before the memory was freed. The block behind `BOOST_SCOPE_EXIT` got a

reference to **i** and freed the memory. But since the block is executed at the end of `foo()`, the assignment of 0 to **i** is too late. The return value of `foo()` is a copy of **i** that gets created before **i** is set to 0.

You can ignore Boost.ScopeExit if you use a C++11 development environment. In that case, you can use RAII without resource-specific classes with the help of lambda functions.

Example 3.2. Boost.ScopeExit with C++11 lambda functions

```
#include <iostream>
#include <utility>

template <typename T>
struct scope_exit
{
  scope_exit(T &&t) : t_{std::move(t)} {}
  ~scope_exit() { t_(); }
  T t_;
};

template <typename T>
scope_exit<T> make_scope_exit(T &&t) { return scope_exit<T>{
  std::move(t)}; }

int *foo()
{
  int *i = new int{10};
  auto cleanup = make_scope_exit([&i]() mutable { delete i; i = 0; });
  std::cout << *i << '\n';
  return i;
}

int main()
{
  int *j = foo();
  std::cout << j << '\n';
}
```

Example 3.2 defines the class `scope_exit` whose constructor accepts a function. This function is called by the destructor. Furthermore, a helper function, `make_scope_exit()`, is defined that makes it possible to instantiate `scope_exit` without having to specify a template parameter.

In `foo()` a lambda function is passed to `make_scope_exit()`. The lambda function looks like the block after `BOOST_SCOPE_EXIT` in Example 3.1: The dynamically allocated `int` variable whose address is stored in **i** is freed with `delete`. Then 0 is assigned to **i**.

The example does the same thing as the previous one. Not only is the `int` variable deleted, but **j** is not set to 0 either when it is written to the standard output stream.

Example 3.3. Peculiarities of `BOOST_SCOPE_EXIT`

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

struct x
{
  int i;

  void foo()
  {
```

```
    i = 10;
    BOOST_SCOPE_EXIT(void)
    {
      std::cout << "last\n";
    } BOOST_SCOPE_EXIT_END
    BOOST_SCOPE_EXIT(this_)
    {
      this_->i = 20;
      std::cout << "first\n";
    } BOOST_SCOPE_EXIT_END
  }
};

int main()
{
  x obj;
  obj.foo();
  std::cout << obj.i << '\n';
}
```

Example 3.3 introduces some peculiarities of `BOOST_SCOPE_EXIT`:

- When `BOOST_SCOPE_EXIT` is used to define more than one block in a scope, the blocks are executed in reverse order. Example 3.3 displays `first` followed by `last`.
- If no variables will be passed to `BOOST_SCOPE_EXIT`, you need to specify `void`. The parentheses must not be empty.
- If you use `BOOST_SCOPE_EXIT` in a member function and you need to pass a pointer to the current object, you must use **`this_`**, not `this`.

Example 3.3 displays `first`, `last`, and `20` in that order.

Exercise

Replace `std::unique_ptr` and the user-defined deleter with `BOOST_SCOPE_EXIT`:

```
#include <string>
#include <memory>
#include <cstdio>

struct CloseFile
{
    void operator()(std::FILE *file)
    {
        std::fclose(file);
    }
};

void write_to_file(const std::string &s)
{
    std::unique_ptr<std::FILE, CloseFile> file{
      std::fopen("hello-world.txt", "a") };
    std::fprintf(file.get(), s.c_str());
}

int main()
{
    write_to_file("Hello, ");
    write_to_file("world!");
}
```