P0052 - Generic Scope Guard and RAII Wrapper for the Standard Library

Peter Sommerlad and Andrew L. Sandoval

2015-09-15

Document Number: P0052	(update of N4189, N3949, N3830, N3677)
Date:	2015-09-15
Project:	Programming Language C++

1 History

1.1 Changes from N4189

- Corrections based on committee feedback.

1.2 Changes from N3949

- renamed `scope_guard` to `scope_exit` and the factory to `make_scope_exit`. Reason for make_ is to teach users to save the result in a local variable instead of just have a temporary that gets destroyed immediately. Similarly for unique resources, `unique_resource`, `make_unique_resource` and `make_unique_resource_checked`.

- renamed editorially `scope_exit::deleter` to `scope_exit::exit_function`.

- changed the factories to use forwarding for the `deleter`/`exit_function` but not deduce a reference.

- get rid of `invoke`'s parameter and rename it to `reset()` and provide a `noexcept` specification for it.

1.3 Changes from N3830

- rename to `unique_resource_t` and factory to `unique_resource`, resp. `unique_resource_checked`
- provide scope guard functionality through type `scope_guard_t` and `scope_guard` factory
- remove multiple-argument case in favor of simpler interface, lambda can deal with complicated release APIs requiring multiple arguments.
- make function/functor position the last argument of the factories for lambda-friendliness.

1.4 Changes from N3677

- Replace all 4 proposed classes with a single class covering all use cases, using variadic templates, as determined in the Fall 2013 LEWG meeting.
- The conscious decision was made to name the factory functions without "make", because they actually do not allocate any resources, like `std::make_unique` or `std::make_shared` do

2 Introduction

The Standard Template Library provides RAII classes for managing pointer types, such as `std::unique_ptr` and `std::shared_ptr`. This proposal seeks to add a two generic RAII wrappers classes which tie zero or one resource to a clean-up/completion routine which is bound by scope, ensuring execution at scope exit (as the object is destroyed) unless released early or in the case of a single resource: executed early or returned by moving its value.

3 Acknowledgements

- This proposal incorporates what Andrej Alexandrescu described as scope_guard long ago and explained again at C++ Now 2012 ().
- This proposal would not have been possible without the impressive work of Peter Sommerlad who produced the sample implementation during the Fall 2013 committee meetings in Chicago. Peter took what Andrew Sandoval produced for N3677 and demonstrated the possibility of using C++14 features to make a single, general purpose RAII wrapper capable of fulfilling all of the needs presented by the original 4 classes (from N3677) with none of the compromises.

- Gratitude is also owed to members of the LEWG participating in the February 2014 (Issaquah) and Fall 2013 (Chicago) meeting for their support, encouragement, and suggestions that have led to this proposal.

- Special thanks and recognition goes to OpenSpan, Inc. (http://www.openspan.com) for supporting the production of this proposal, and for sponsoring Andrew L. Sandoval's first proposal (N3677) and the trip to Chicago for the Fall 2013 LEWG meeting. *Note: this version abandons the over-generic version from N3830 and comes back to two classes with one or no resource to be managed.*

- Thanks also to members of the mailing lists who gave feedback. Especially Zhihao Yuan, and Ville Voutilainen.

- Special thanks to Daniel Krügler for his deliberate review of the draft version of this paper (D3949).

4 Motivation and Scope

The quality of C++ code can often be improved through the use of "smart" holder objects. For example, using `std::unique_ptr` or `std::shared_ptr` to manage pointers can prevent common mistakes that lead to memory leaks, as well as the less common leaks that occur when exceptions unwind. The latter case is especially difficult to diagnose and debug and is a commonly made mistake – especially on systems where unexpected events (such as access violations) in third party libraries may cause deep unwinding that a developer did not expect. (One example would be on Microsoft Windows with Structured Exception Handling and libraries like MFC that issue callbacks to user-defined code wrapped in a `try/catch(...)` block. The developer is usually unaware that their code is wrapped with an exception handler that depending on compile-time options will quietly unwind their code, masking any exceptions that occur.)

While `std::unique_ptr` can be tweaked by using a custom deleter type to almost a perfect handler for resources, it is awkward to use for handle types that are not pointers and for the use case of a scope guard. As a smart pointer `std::unique_ptr` can be used syntactically like a pointer, but requires the use of `get()` to pass the underlying pointer value to legacy APIs.

This proposal introduces two new RAII "smart" resource containers. The first is called `unique_resource` which can bind a resource to "clean-up" code regardless of the type of the argument required by the "clean-up" function. The second is called `scope_exit` which can bind a parameter-less function or lambda to scope exit, allowing clean-up of a resource requiring zero (parameter-less) or more (via lambda capture) variables or parameters.

4.1 Without Coercion

Existing smart pointer types can often be coerced into providing the needed functionality. For example, `std::unique_ptr` could be coerced into invoking a function used to close an opaque handle type. For example, given the following system APIs, `std::unique_ptr` can be used to ensure the file handle is not leaked on scope exit:

```
typedef void *HANDLE;          // System defined opaque handle type
typedef unsigned long DWORD;
#define INVALID_HANDLE_VALUE reinterpret_cast<HANDLE>(-1)
// Can't help this, that's from the OS

// System defined APIs
void CloseHandle(HANDLE hObject);

HANDLE CreateFile(const char *pszFileName,
        DWORD dwDesiredAccess,
        DWORD dwShareMode,
        DWORD dwCreationDisposition,
        DWORD dwFlagsAndAttributes,
        HANDLE hTemplateFile);

bool ReadFile(HANDLE hFile,
        void *pBuffer,
        DWORD nNumberOfBytesToRead,
        DWORD*pNumberOfBytesRead);

// Using std::unique_ptr to ensure file handle is closed on scope-exit:
void CoercedExample()
{
        // Initialize hFile ensure it will be "closed" (regardless of value) on scope-exit
        std::unique_ptr<void, decltype(&CloseHandle)> hFile(
                CreateFile("test.tmp",
                        FILE_ALL_ACCESS,
                        FILE_SHARE_READ,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        nullptr),
                CloseHandle);

        // Read some data using the handle
        std::array<char, 1024> arr = { };
        DWORD dwRead = 0;
        ReadFile(hFile.get(),    // Must use std::unique_ptr::get()
                &arr[0],
                static_cast<DWORD>(arr.size()),
                &dwRead);
}
```

While this works, there are a few problems with coercing `std::unique_ptr` into handling the resource in this manner:

- The type used by the `std::unique_ptr` does not match the type of the resource. `void` is not a `HANDLE`. (Thus the word coercion is used to describe it.)
- There is no convenient way to check the value returned by `CreateFile` and assigned to the `std::unique_ptr<void>` to prevent calling `CloseHandle` when an invalid handle value is returned. `std::unique_ptr` will check for a null pointer, but the `CreateFile` API may return another pre-defined value to signal an error.
- Because hFile does not have a cast operator that converts the contained "pointer" to a `HANDLE`, the `get()` method must be used when invoking other system APIs needing the underlying `HANDLE`.

Each of these problems is solved by `unique_resource` as shown in the following example:

```
void ScopedResourceExample1()
{
        // Initialize hFile ensure it will be "closed" (regardless of value) on scope-exit
        auto hFile = std::make_unique_resource_checked(
                CreateFile("test.tmp",
                        FILE_ALL_ACCESS,
                        FILE_SHARE_READ,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        nullptr),           // The resource
                INVALID_HANDLE_VALUE,   // Don't call CloseHandle if it failed!
                CloseHandle);           // Clean-up API, lambda-friendly position

        // Read some data using the handle
        std::array<char, 1024> arr = { };
        DWORD dwRead = 0;
        // cast operator makes it seamless to use with other APIs needing a HANDLE
        ReadFile(hFile,
                &arr[0],
                static_cast<DWORD>(arr.size()),
                &dwRead);
}
```

4.1.1 Non-Pointer Handle Types

While `std::unique_ptr` can deal with the above pointer handle type, as well as `<cstdio>`'s `FILE *`, it is non-intuitive to use with handle's like `<fcntl.h>`'s and `<unistd.h>`'s `int` file handles. See the following code examples on using `unique_resource` with `int` and `FILE *` handle types.

```
void demonstrate_unique_resource_with_stdio()
{
        const std::string filename = "hello.txt";
        {
                auto file=make_unique_resource(::fopen(filename.c_str(),"w"),&::fclose);
                ::fputs("Hello World!\n", file);
                ASSERT(file.get()!= NULL);
        }
        {
                std::ifstream input { filename.c_str() };
                std::string line { };
                getline(input, line);
                ASSERT_EQUAL("Hello World!", line);
                getline(input, line);
                ASSERT(input.eof());
        }
        ::unlink(filename.c_str());
        {
                auto file = make_unique_resource_checked(::fopen("nonexistingfile.txt", "r")
                            (FILE*) NULL, &::fclose);
                ASSERT_EQUAL((FILE*)NULL, file.get());
        }

}
```

```
void demontrate_unique_resource_with_POSIX_IO()
{
        const std::string filename = "./hello1.txt";
        {
                auto file=make_unique_resource(::open(filename.c_str(),
                     O_CREAT|O_RDWR|O_TRUNC,0666), &::close);

                ::write(file, "Hello World!\n", 12u);
                ASSERT(file.get() != -1);
        }
        {
                std::ifstream input { filename.c_str() };
                std::string line { };
                getline(input, line);
                ASSERT_EQUAL("Hello World!", line);
                getline(input, line);
                ASSERT(input.eof());
        }
        ::unlink(filename.c_str());
        {
                auto file = make_unique_resource_checked(::open("nonexistingfile.txt",
                       O_RDONLY), -1, &::close);
                ASSERT_EQUAL(-1, file.get());
        }

}
```

4.2 Multiple Parameters

This feature was abandoned due to feedback by LEWG in Issaquah. A lambda as an exit function can have the same effect without complicating `unique_resource` or `scope_exit`.

4.3 Lambdas, multiple parameters, and zero parameters. with `scope_exit`

While `unique_resource` is ideal for wrapping resources which are cleaned-up when the resource is passed to a clean-up function taking a single parameter of the type of the resource, `scope_exit` may be used in cases where the clean-up function requires zero or multiple parameters, as shown in the following example:

```
void ExampleOfMultipleAndZeroParameterCleanup()
{
        // Initialize COM calls and ensure CoUninitialize() is called before leaving scope
        // Demonstrates a zero-parameter clean-up call
        CoInitialize(nullptr);
        auto aUninitialize = make_scope_exit(std::ref(CoUninitialize));

        // Allocate a block of virtual memory with execute permissions
        // Make sure it is cleaned up on scope-exit.
        // The clean-up function (VirtualFree) requires 3 parameters...
        void *pvExecutionChamber = VirtualAlloc(nullptr, executableCode.size(),
                MEM_COMMIT,
                PAGE_EXECUTE_READWRITE);
        auto aCleanupVM = make_scope_exit([pvExecutionChamber]() ->void)
        {
                if(nullptr != pvExecutionChamber)
                        VirtualFree(pvExecutionChamber, 0, MEM_RELEASE);
        });

        // Do other stuff here with the above resources...
}
```

It is also possible to release a `scope_exit` instance in order to prevent the clean-up code from running as can be seen in the following (otherwise useless) example:

```
void TalkToTheWorld(std::ostream& out, std::string const farewell="Uff Wiederluege...")
{
        // Always say goodbye before returning,
        // but if given a non-empty farewell message use it...
        auto goodbye = make_scope_exit([&out]() ->void
        {
                out << "Goodbye world..." << std::endl;
        });
        auto altgoodbye = make_scope_exit([&out,farewell]() ->void
```

```
        {
                out << farewell << std::endl;
        });


        if(farewell.empty())
        {
                altgoodbye.release();           // Don't use farewell!
        }
        else
        {
                goodbye.release();      // Don't use the alternate
        }
}


void testTalkToTheWorld()
{
        std::ostringstream out;
        TalkToTheWorld(out,"");
        ASSERT_EQUAL("Goodbye world...\n",out.str());
        out.str("");
        TalkToTheWorld(out);
        ASSERT_EQUAL("Uff Wiederluege...\n",out.str());
}
```

4.4 Other Functionality

In addition to the basic features shown above, `unique_resource` also provides various operators (cast, `->`, `()`, and accessor methods (`get`, `get_deleter`). The most complicated of these is the `reset()` member function which allows the "clean-up" function to be executed early, just as it would be at scope exit. The `reset(R&& resource)` member function also allows the resource value to be reset, causing clean-up of the previously owned resource unless it had been released. The newly assigned resource will then be cleaned up at scope exit.

As already shown in the examples, the expected method of construction for non-member variables is to use one of the two generator functions:

- `unique_resource(resources,deleter)` - non-checking instance.
- `unique_resource_checked(resource, invalid_value,deleter)` - checked instance, allowing a resource which is validated to inhibit the call to the deleter function if invalid.

4.5 What's not included

`unique_resource` does not do reference counting like `shared_ptr` does. Though there is very likely a need for a class similar to `unique_resource` that includes reference counting it is beyond the scope of this proposal.

One other limitation with `unique_resource` and `scope_exit` is that while the resources themselves may be `reset()`, the "deleter" or "clean-up" function/lambda can not be altered, because they are part of the type. Generally there should be no need to reset the deleter, and especially with lambdas type matching would be difficult or impossible.

5 Impact on the Standard

This proposal is a pure library extension. Two new headers, `<scope_exit>` and `<unique_resource>` are proposed, but it does not require changes to any standard classes or functions. It does not require any changes in the core language, and it has been implemented in standard C++ conforming to C++14. Depending on the timing of the acceptance of this proposal, it might go into library fundamentals TS under the namespace std::experimental or directly in the working paper of the standard, once it is open again for future additions.

6 Design Decisions

6.1 General Principles

The following general principles are formulated for `unique_resource`, and are valid for `scope_exit` correspondingly.

- Simplicity - Using `unique_resource` should be nearly as simple as using an unwrapped type. The generator functions, cast operator, and accessors all enable this.
- Transparency - It should be obvious from a glance what each instance of a `unique_resource` object does. By binding the resource to it's clean-up routine, the declaration of `unique_resource` makes its intention clear.
- Resource Conservation and Lifetime Management - Using `unique_resource` makes it possible to "allocate it and forget about it" in the sense that deallocation is always accounted for after the `unique_resource` has been initialized.
- Exception Safety - Exception unwinding is one of the primary reasons that `unique_resource` is needed. Nevertheless the goal is to introduce a new container that will not throw during construction of the `unique_resource` itself. However, there are no intentions to provide safeguards for piecemeal construction of resource and

deleter. If either fails, no unique_resource will be created, because the factory function unique_resource will not be called. It is not recommended to use `unique_resource()` factory with resource construction, functors or lambda capture types where creation, copying or moving might throw.

- Flexibility - `unique_resource` is designed to be flexible, allowing the use of lambdas or existing functions for clean-up of resources.

6.2 Prior Implementations

Please see N3677 from the May 2013 mailing (or http://www.andrewlsandoval.com/scope_exit/) for the previously proposed solution and implementation. Discussion of N3677 in the (Chicago) Fall 2013 LEWG meeting led to the creation of `unique_resource` and `scope_exit` with the general agreement that such an implementation would be vastly superior to N3677 and would find favor with the LEWG. Professor Sommerlad produced the implementation backing this proposal during the days following that discussion.

N3677 has a more complete list of other prior implementations.

N3830 provided an alternative approach to allow an arbitrary number of resources which was abandoned due to LEWG feedback

The following issues have been discussed by LEWG already:

- *Should there be a companion class for sharing the resource* `shared_resource` *? (Peter thinks no. Ville thinks it could be provided later anyway.)* LEWG: NO.
- *Should* `scope_exit()` *and* `unique_resource::invoke()` *guard against deleter functions that throw with* `try deleter(); catch(...)` *(as now) or not?* LEWG: NO, but provide noexcept in detail.
- *Does* `scope_exit` *need to be move-assignable?* LEWG: NO.

6.3 Open Issues to be Discussed

- Should we make the regular constructors private and friend the factory functions only?
- Should we provide a factory for type-erasing the deleter/exit_function using std::function?

7 Technical Specifications

The following formulation is based on inclusion to the draft of the C++ standard. However, if it is decided to go into the Library Fundamentals TS, the position of the texts and the namespaces will have to be adapted accordingly, i.e., instead of namespace `std::` we suppose namespace `std::experimental::`.

7.1 Header

In section [utilities.general] add two extra rows to table 44

Table 1: Table 44 - General utilities library summary

Subclause	Header
20.nn Scope Guard Support	`<scope_exit>`
20.nn+1 Unique Resource Wrapper	`<unique_resource>`

7.2 Additional sections

Add a two new sections to chapter 20 introducing the contents of the headers `<scope_exit>` and `<unique_resource>`.

7.3 Scope Guard Support [utilities.scope_exit]

This subclause contains infrastructure for a generic scope guard.

Header `<scope_exit>` synopsis

1 The header `<scope_exit>` defines the class template `scope_exit` and the function template `make_scope_exit()` to create its instances.

```
namespace std {
template <typename EF>
struct scope_exit {
        // construction
        explicit
        scope_exit(EF &&f) noexcept;
        // move
        scope_exit(scope_exit  &&rhs) noexcept;
        // release
        ~scope_exit() noexcept(noexcept(this->exit_function()));
        void release() noexcept;

private:
        scope_exit(scope_exit const &)=delete;
        scope_exit& operator=(scope_exit const &)=delete;
        scope_exit& operator=(scope_exit &&)=delete;
        EF exit_function;               // exposition only
        bool execute_on_destruction;    // exposition only
};
// factory function
template <typename EF>
```

```
auto make_scope_exit(EF &&exit_function) noexcept;

} // namespace std
```

2 [*Note:* `scope_exit` is meant to be a universal scope guard to call its deleter function on scope exit. — *end note*]

7.3.1 Class Template `scope_exit` [scope_exit.scope_exit]

1 *Requires:* `EF` shall be a MoveConstructible function object type or reference to such. The expression `exit_function()` shall be valid. Move construction of `EF` shall not throw an exception.

```
explicit
scope_exit(EF &&exit_function) noexcept;
```

2 *Effects:* constructs a scope_exit object that will call `exit_function()` on its destruction if not `release()`ed prior to that.

```
~scope_exit();
```

3 *Effects:* Calls `exit_function()` unless `release()` was previously called.

```
void release() noexcept;
```

4 *Effects:* Prevents `exit_function()` from being called on destruction.

```
scope_exit(scope_exit  &&rhs) noexcept;
```

5 *Effects:* Move constructs `exit_function` from `rhs.exit_function`. Copies the release state from rhs, and sets rhs to the released state, preventing it from invoking its copy of `exit_function`.

7.3.2 Factory Function `make_scope_exit` [scope_exit.make_scope_exit]

```
template <typename EF>
scope_exit<remove_reference_t<EF>> make_scope_exit(EF && exit_function) noexcept;
```

1 *Returns:* `scope_exit<std::remove_reference_t<EF>>(std::forward<EF>(exit_function))`

7.4 Unique Resource Wrapper [utilities.unique_resource]

This subclause contains infrastructure for a generic RAII resource wrapper.

Header `<unique_resource>` synopsis

1 The header `<unique_resource>` defines the class template `unique_resource` and function templates `make_unique_resource()` and `make_unique_resource_checked()` to create its instances.

```
namespace std {

template<typename R,typename D>
class unique_resource {
        R resource; // exposition only
```

```
        D deleter; // exposition only
        bool execute_on_destruction; // exposition only
        unique_resource& operator=(unique_resource const &)=delete;
        unique_resource(unique_resource const &)=delete;
public:
        // construction
        explicit
        unique_resource(R && resource, D && deleter, bool shouldRun=true) noexcept;
        // move
        unique_resource(unique_resource &&other) noexcept;
        unique_resource& operator=(unique_resource  &&other) noexcept ;

        // resource release
        ~unique_resource() noexcept(noexcept(this->reset()));
        void reset() noexcept(noexcept(this->get_deleter()(resource)));
        void reset(R && newresource) noexcept(noexcept(this->reset())) ;
        R const & release() noexcept;
        // resource access
        R const & get() const noexcept ;
        operator  R const &() const noexcept ;
        R operator->() const noexcept ;
        // deleter access
        const D &       get_deleter() const noexcept;
};

//factories
template<typename R,typename D>
unique_resource<R,remove_reference_t<D>>
make_unique_resource( R && r,D &&d) noexcept;

template<typename R,typename D>
unique_resource<R,D>
make_unique_resource_checked(R r, R invalid, D d) noexcept;

} // namespace std
```

[2] [*Note:* `unique_resource` is meant to be a universal RAII wrapper for resource handles provided by an operating system or platform. Typically, such resource handles come with a factory function and a clean-up or deleter function and are of trivial type. The clean-up function together with the result of the factory function is used to create a `unique_resource` variable, that on destruction will call the clean-up function. Access to the underlying resource handle is achieved through a set of convenience functions or type conversion. — *end note*]

7.4.1 Class Template `unique_resource` [unique_resource.unique_resource]

1 *Requires:* `D` and `R` shall be a MoveConstructible and MoveAssignable. `D` shall be a function object type or reference to such. The expression `deleter(resource)` shall be valid. Move construction and move assignment of `D` and `R` shall not throw an exception.

```
explicit
unique_resource(R && resource, D && deleter, bool shouldRun=true) noexcept;
```

2 *Effects:* constructs a `unique_resource` by moving `resource` and then `deleter`. The constructed object will call `deleter(resource)` on its destruction if not `release()`ed prior to that. On construction the resource is to be in a non-released state.

```
unique_resource(unique_resource &&other) noexcept;
```

3 *Effects:* move-constructs a `unique_resource` from `other`'s members then calls `other.release()`.

```
unique_resource& operator=(unique_resource  &&other) noexcept ;
```

4 *Effects:* `this->reset();` Move-assigns members from `other` then calls `other.release()`.

```
~unique_resource() ;
```

5 *Effects:* `this->reset();`

```
void reset() noexcept(noexcept(this->get_deleter()(resource)));
```

6 *Effects:* If `release()` has not been called, invokes the equivalent of `this->get_deleter()(resource);` Otherwise no action is taken.

```
void reset(R && newresource) noexcept ;
```

7 *Effects:* Invokes the deleter function for resource if it was not previously released, e.g. `this->reset();` Then moves newresource into the tracked resource member, e.g. `this->resource = std::move(newresource);` Finally sets the object in the non-released state so that the deleter function will be invoked on destruction if `release()` is not called first.

8 [*Note:* This function takes the role of an assignment of a new resource. — *end note*]

```
R const & release() noexcept;
```

9 *Effects:* Set the object in the released state so that the deleter function will not be invoked on destruction or `reset()`.

10 *Returns:* `resource`

```
R const & get() const noexcept ;
operator  R const &() const noexcept ;
R operator->() const noexcept ;
```

11 *Requires:* `operator->` is only available if `is_pointer<R>::value && (is_class<remove_pointer_t<R>>::value || is_union<remove_pointer_t<R>>::value)` is `true`.

12 *Returns:* `resource`.

```
const DELETER & get_deleter() const noexcept;
```
13 *Returns:* `deleter`

7.4.2 Factories for `unique_resource` [unique_resource.unique_resource]

```
template<typename R,typename D>
unique_resource<R,remove_reference_t<D>>
make_unique_resource( R && r,D &&d) noexcept;
```
1 *Returns:* `unique_resource<R,remove_reference_t<D>>(std::move(r), std::forward<remove_reference_t<D>>(d),true)`

```
template<typename R,typename D>
unique_resource<R,D>
make_unique_resource_checked(R r, R invalid, D d ) noexcept;
```
2 *Requires:* `R` is EqualityComparable

3 *Returns:* `unique_resource<R,D>(std::move(r), std::move(d), not bool(r==invalid))`

8 Appendix: Example Implementations

8.1 Scope Guard Helper

```
#ifndef SCOPE_EXIT_H_
#define SCOPE_EXIT_H_

// modeled slightly after Andrescu's talk and article(s)

namespace std{
namespace experimental{

template <typename EF>
struct scope_exit {
        // construction
        explicit
        scope_exit(EF &&f) noexcept
        :exit_function(std::move(f))
        ,execute_on_destruction{true}{ }
        // move
        scope_exit(scope_exit  &&rhs) noexcept
        :exit_function(std::move(rhs.exit_function))
        ,execute_on_destruction{rhs.execute_on_destruction}{
                rhs.release();
        }
        // release
        ~scope_exit() noexcept(noexcept(this->exit_function())){
                if (execute_on_destruction)
                                this->exit_function();
```

```
        }
        void release() noexcept { this->execute_on_destruction = false;}

private:
        scope_exit(scope_exit const &)=delete;
        void operator=(scope_exit const &)=delete;
        scope_exit& operator=(scope_exit &&)=delete;
        EF exit_function;
        bool execute_on_destruction; // exposition only
};

template <typename EF>
auto make_scope_exit(EF &&exit_function) noexcept {
        return scope_exit<std::remove_reference_t<EF>>(std::forward<EF>(exit_function));
}

}
}

#endif /* SCOPE_EXIT_H_ */
```

8.2 Unique Resource

```
#ifndef UNIQUE_RESOURCE_H_
#define UNIQUE_RESOURCE_H_

namespace std{
namespace experimental{

template<typename R,typename D>
class unique_resource{
        R resource;
        D deleter;
        bool execute_on_destruction; // exposition only
        unique_resource& operator=(unique_resource const &)=delete;
        unique_resource(unique_resource const &)=delete; // no copies!
public:
        // construction
        explicit
        unique_resource(R && resource, D && deleter, bool shouldrun=true) noexcept
                :  resource(std::move(resource))
                ,  deleter(std::move(deleter))
                , execute_on_destruction{shouldrun}{}
        // move
        unique_resource(unique_resource &&other) noexcept
        :resource(std::move(other.resource))
        ,deleter(std::move(other.deleter))
        ,execute_on_destruction{other.execute_on_destruction}{
                other.release();
```

```
        }
        unique_resource&
        operator=(unique_resource  &&other) noexcept(noexcept(this->reset())) {
                this->reset();
                this->deleter=std::move(other.deleter);
                this->resource=std::move(other.resource);
                this->execute_on_destruction=other.execute_on_destruction;
                other.release();
                return *this;
        }
    // resource release
        ~unique_resource() noexcept(noexcept(this->reset())){
                this->reset();
        }
        void reset() noexcept(noexcept(this->get_deleter()(resource))) {
                if (execute_on_destruction) {
                        this->execute_on_destruction = false;
                        this->get_deleter()(resource);
                }
        }
        void reset(R && newresource) noexcept(noexcept(this->reset())) {
                this->reset();
                this->resource = std::move(newresource);
                this->execute_on_destruction = true;
        }
        R const & release() noexcept{
                this->execute_on_destruction = false;
                return this->get();
        }

        // resource access
        R const & get() const noexcept {
                return this->resource;
        }
        operator  R const &() const noexcept {
                return this->resource;
        }
        R
        operator->() const noexcept {
                return this->resource;
        }

        // deleter access
        const D &
        get_deleter() const noexcept {
                return this->deleter;
        }
};
```

```
//factories
template<typename R,typename D>
auto
make_unique_resource( R && r,D &&d) noexcept {
        return unique_resource<R,std::remove_reference_t<D>>(
                        std::move(r)
                        ,std::forward<std::remove_reference_t<D>>(d)
                        ,true);
}

template<typename R,typename D>
auto
make_unique_resource_checked(R r, R invalid, D d ) noexcept {
        bool shouldrun = not bool(r == invalid);
        return unique_resource<R,D>(std::move(r), std::move(d), shouldrun);
}



}}
#endif /* UNIQUE_RESOURCE_H_ */
```