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

Peter Sommerlad and Andrew L. Sandoval

2015-09-27

| Document Number: P0052 | (update of N4189, N3949, N3830, N3677) |
|---|---|
| Date: | 2015-09-27 |
| Project: | Programming Language C++ |

# 1 History

## 1.1 Changes from N4189

- Attempt to address LWG specification issues from Cologne (only learned about those in the week before the deadline from Ville, so not all might be covered).
  - specify that the exit function must be either no-throw copy-constructible, or no-throw move-constructible, or held by reference. Stole the wording and implementation from unique_ptr's deleter ctors.
  - put both classes in single header `<scope>`
  - specify factory functions for Alexandrescu's 3 scope exit cases for `scope_exit`. Deliberately did't provide similar things for `unique_resource`.
- remove lengthy motivation and example code, to make paper easier digestible.
- Corrections based on committee feedback in Urbana and Cologne.

## 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 Impact on the Standard

This proposal is a pure library extension. A new headers, `<scope>` is 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.

# 5 Design Decisions

## 5.1 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.
- Alexandrescu's `SCOPE_SUCCESS` and `SCOPE_FAIL` macros that make use of the new `uncaught_exceptions()` functionality are not (yet) in the scope of this suggestions, but could be added easily through additional factory functions and a possible additional template parameter.

## 5.2 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.

## 5.3 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?
- Should we provide factories `make_scope_success(ef)` and `make_scope_fail(ef)` to enable Alexandrescu's three scope-exiting modes?

# 6 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::`.

## 6.1 Header

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

Table 1: Table 44 - General utilities library summary

| | Subclause | Header |
|---|---|---|
| 20.nn | Scope Guard Support | `<scope>` |

## 6.2 Additional sections

Add a a new section to chapter 20 introducing the contents of the header `<scope>`.

## 6.3 Scope Guard Support [utilities.scope]

This subclause contains infrastructure for a generic scope guard and RAII resource wrapper.

**Header `<scope>` synopsis**

```
namespace std {
template <typename EF>
struct scope_exit;

template <typename EF>
```

```
scope_exit<see below> make_scope_exit(EF &&exit_function) noexcept;
template <typename EF>
scope_exit<see below> make_scope_fail(EF && exit_function) noexcept;
template <typename EF>
scope_exit<see below> make_scope_success(EF && exit_function) noexcept;

template<typename R,typename D>
class unique_resource;

template<typename R,typename D>
unique_resource<R,see below>
make_unique_resource( R && r,D&& d) noexcept;

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

}
```

1 The header `<scope>` defines the class templates `scope_exit` `unique_resource` and the factory function templates `make_scope_exit()`, `make_scope_success()`, `make_scope_fail()`, `make_unique_resource` and `make_unique_resource_checked` to create their instances. The usage of the RAII wrappers assumes that the exit functions/deleter provided do not throw exceptions and that they are either `nothrow_move_constructible` or `nothrow_copy_constructible` or kept by reference. [*Note:* If a user desires type erasure on the scope exit functions, wrapping in `std::function` is recommended. — *end note*]

### 6.3.1 Class Template `scope_exit` [scope.scope_exit]

```
template <typename EF>
struct scope_exit {
        // construction
        explicit
        scope_exit(see below f1) noexcept;
        explicit
        scope_exit(see below f2) noexcept;
        // move
        scope_exit(scope_exit  &&rhs) noexcept;
        // release
        ~scope_exit() ;
        void release() noexcept;

        scope_exit(scope_exit const &)=delete;
        scope_exit& operator=(scope_exit const &)=delete;
        scope_exit& operator=(scope_exit &&)=delete;
private:
        EF exit_function;               // exposition only
```

```
};
```

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

2 A client-supplied template argument `EF` shall be a function object type ( 20.9 ), lvalue reference to function, or lvalue reference to function object type for which, given a value `f` of type `EF` the expression `f()` is valid.

3 *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.

4 If the exit function type `EF` is not a reference type, `EF` shall satisfy the requirements of `Destructible` (Table 24 ).

```
explicit
scope_exit(see below f1) noexcept;
explicit
scope_exit(see below f2) noexcept;
```

5 The signature of these constructors depends upon whether `EF` is a reference type. If `EF` is non-reference type `A`, then the signatures are:

```
scope_exit(const A& f);
scope_exit(A&& f);
```

6 If `EF` is an lvalue reference type `A&`, then the signatures are:

```
scope_exit(A& f);
scope_exit(A&& f);
```

7 If `EF` is an lvalue reference type `const A&`, then the signatures are:

```
scope_exit(const A& f);
scope_exit(const A&& f);
```

8 *Requires:*

- If `EF` is not an lvalue reference type then
  - If `f` is an lvalue or `const` rvalue then the first constructor of this pair will be selected. `EF` shall satisfy the requirements of `CopyConstructible` (Table 21 ), and the copy constructor of `EF` shall not throw an exception. This `scope_exit` will hold a copy of `f`.
  - Otherwise, `f` is a non-const rvalue and the second constructor of this pair will be selected. `EF` shall satisfy the requirements of `MoveConstructible` (Table 20 ), and the move constructor of `EF` shall not throw an exception. This `scope_exit` will hold a value move constructed from `f`.
- Otherwise `EF` is an lvalue reference type. `f` shall be reference-compatible with one of the constructors. If `f` is an rvalue, it will bind to the second constructor of this pair and the program is ill-formed. [ *Note:* The diagnostic could be

implemented using a `static_assert` which assures that `EF` is not a reference type. — *end note* ] Else `f` is an lvalue and will bind to the first constructor of this pair. The type which `EF` references need not be `CopyConstructible` nor `MoveConstructible`. This `scope_exit` will hold a `EF` which refers to the lvalue `f`. [ *Note:* `EF` may not be an rvalue reference type. — *end note* ]

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

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

10 *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`.

```
~scope_exit();
```

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

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

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

### 6.3.2 `scope_exit` Factory Functions [scope.make_scope_exit]

```
template <typename EF>
scope_exit<see below> make_scope_exit(EF && exit_function) noexcept;
template <typename EF>
scope_exit<see below> make_scope_fail(EF && exit_function) noexcept;
template <typename EF>
scope_exit<see below> make_scope_success(EF && exit_function) noexcept;
```

1 The factory functions create `scope_exit` objects, that run `exit_function` at scope exit if not `release()`d under the following conditions:

**make_scope_exit** always, if scope is left

**make_scope_fail** if scope is left with an exception

**make_scope_success** if scope is left without any exception

2 [ *Note:* These factory functions allow declarative control flow when used with lambdas as arguments as introduced by Andrej Alexandrescu. If the `exit_function` throws when called from `scope_exit`'s destructor and that object was not constructed with `make_scope_success`, this causes the program to `terminate()`. — *end note* ]

3 The return value is as follows: If EF is a non-const lvalue reference type

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

5 otherwise

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

7 [ *Note:* The first case keeps the exit function by reference the other by value. To enable type erasure for `scope_exit` objects, e.g., for keeping them as class members, use `function<void()>` as template argument. — *end note* ]

### 6.3.3 Unique Resource Wrapper [scope.unique_resource]

### 6.3.4 Class Template `unique_resource` [scope.unique_resource.unique_resource]

```
template<typename R,typename D>
class unique_resource {
        R resource; // exposition only
        D deleter; // exposition only
        bool execute_on_destruction; // exposition only
public:
        // construction
        unique_resource(R && resource, see below d1, bool shouldRun=true) noexcept;
        unique_resource(R && resource, see below d2, bool shouldRun=true) noexcept;
        // move
        unique_resource(unique_resource &&other) noexcept;
        unique_resource& operator=(unique_resource  &&other) noexcept ;

    // resource release
        ~unique_resource() ;
        void reset() ;
        void reset(R && newresource) ;
        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;

        unique_resource& operator=(unique_resource const &)=delete;
        unique_resource(unique_resource const &)=delete;
};
```

1 [ *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 are of trivial type and come with a factory function and a clean-up or deleter function that

do not throw exceptions. 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* ]

2 The template argument `D` shall be a function object type (20.9 ), lvalue reference to function, or lvalue reference to function object type for which, given a value `d` of type `D` and a value `r` of type `R`, the expression `d(r)` is valid and does not throw an exception.

3 If the deleter's type `D` is not a reference type, `D` shall be MoveConstructible and satisfy the requirements of `Destructible` (Table 24 ).

4 `R` shall be a MoveConstructible and MoveAssignable. Move construction and move assignment of `D` and `R` shall not throw an exception.

```
unique_resource(R && resource, see below d1, bool shouldRun=true) noexcept;
unique_resource(R && resource, see below d2, bool shouldRun=true) noexcept;
```

5 *Effects:* constructs a `unique_resource` by moving `resource`. 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.

6 The signature of these constructors depends upon whether `D` is a reference type. If `D` is non-reference type `A`, then the signatures are:

```
unique_resource(R&& p, const A& deleter);
unique_resource(R&& p, A&& deleter);
```

7 If `D` is an lvalue reference type `A&`, then the signatures are:

```
unique_resource(R&& p, A& deleter);
unique_resource(R&& p, A&& deleter);
```

8 If `D` is an lvalue reference type `const A&`, then the signatures are:

```
unique_resource(R&& p, const A& deleter);
unique_resource(R&& p, const A&& deleter);
```

9 *Requires:*

- If `D` is not an lvalue reference type then
  - If `deleter` is an lvalue or `const` rvalue then the first constructor of this pair will be selected. `D` shall satisfy the requirements of `CopyConstructible` (Table 21 ), and the copy constructor of `D` shall not throw an exception. This `unique_resource` will hold a copy of `deleter`.
  - Otherwise, `deleter` is a non-const rvalue and the second constructor of this pair will be selected. `D` shall satisfy the requirements of `MoveConstructible` (Table 20 ), and the move constructor of `D` shall not throw an exception. This `unique_resource` will hold a value move constructed from `deleter`.
- Otherwise `D` is an lvalue reference type. `deleter` shall be reference-compatible with one of the constructors. If `deleter` is an rvalue, it will bind to the second constructor of this pair and the program is ill-formed. [ *Note:* The diagnostic

could be implemented using a `static_assert` which assures that `D` is not a reference type. — *end note* ] Else `deleter` is an lvalue and will bind to the first constructor of this pair. The type which `D` references need not be `CopyConstructible` nor `MoveConstructible`. This `unique_resource` will hold a `D` which refers to the lvalue `deleter`. [ *Note:* `D` may not be an rvalue reference type. — *end note* ]

10 *Postconditions:* `get() == r`. `get_deleter()` returns a reference to the stored deleter. If `D` is a reference type then `get_deleter()` returns a reference to the lvalue `deleter`.

[ *Example:*

```
D d;
unique_resource<int, D> p1(42, D());         // D must be MoveConstructible
unique_resource<int, D> p2(42, d);           // D must be CopyConstructible
unique_resource<int, D&> p3(42, d);          // p3 holds a reference to d
unique_resource<int, const D&> p4(42, D()); // error: rvalue deleter object com-
bined
                                             // with reference deleter type
```

— *end example* ]

`unique_resource(unique_resource &&other) noexcept;`

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

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

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

`~unique_resource() ;`

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

`void reset() ;`

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

`void reset(R && newresource) ;`

15 *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.

16 [ *Note:* This function takes the role of an assignment of a new resource. — *end note* ] [ *Note:* If calling the `get_deleter()(get())` might throw, use of one of the `reset()` member functions in a try-block can avoid termination if unique_resource is deleted during stack unwinding when another exception is thrown. Afterwards the unique_resource should be `release()`d before it is destroyed. — *end note* ]

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

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

18 *Returns:* `resource`

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

19 *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`.

20 *Returns:* `resource`.

```
const DELETER & get_deleter() const noexcept;
```

21 *Returns:* `deleter`

### 6.3.5 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 The return value is as follows: If D is a non-const lvalue reference type

2 *Returns:* `unique_resource<R,D>(std::move(r), std::forward<D>(d),true)`

3 otherwise

4 *Returns:* `unique_resource<R,remove_reference_t<D>>(std::move(r), std::forward<D>(d),true)`

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

5 *Requires:* `R` is EqualityComparable

6 The return value is as follows: If D is a non-const lvalue reference type

7 *Returns:* `unique_resource<R,D>(std::move(r), std::forward<D>(d),!bool(r==invalid)`

8 otherwise

9 *Returns:* `unique_resource<R,remove_reference_t<D>>(std::move(r), std::forward<D>(d),!bool(r==invalid)`

# 7 Appendix: Example Implementations

This implementation is incomplete and might not conform to the specification.

## 7.1 Scope Guard Helper

```
#ifndef SCOPE_EXIT_H_
#define SCOPE_EXIT_H_

#include <exception>
// modeled slightly after Andrescu's talk and article(s)

namespace std{
namespace experimental{

namespace _detail{
enum class scope_run{
        exit,success,fail
};
struct exception_counter{
        exception_counter() noexcept
        :count{std::uncaught_exceptions()}
        {
        }
        bool is_new_exception() const {
                return std::uncaught_exceptions() > count;
        }
        int count;
};
}

template <typename EF,_detail::scope_run scope=_detail::scope_run::exit>
struct scope_exit {
        // construction
        explicit
        scope_exit(std::conditional_t<
                std::is_reference<EF>{}
                ,EF
                ,std::add_lvalue_reference_t<EF const>>
                f) noexcept
        :exit_function{f}
        {}
        explicit
        scope_exit(std::remove_reference_t<EF> &&f) noexcept
        :exit_function{std::move(f)}
        {
                static_assert(std::is_nothrow_move_constructible<EF>::value
                        ,"EV must be nothrow move constructible");
```

```
                static_assert(!std::is_lvalue_reference<EF>{}
                        ,"can not pass rvalue for reference type");
        }
        // move
        scope_exit(scope_exit  &&rhs) noexcept
        :exit_function{std::forward<EF>(rhs.exit_function)}
        ,execute_on_destruction_flag{rhs.execute_on_destruction_flag}
        {
                rhs.release();
        }
        // release
        ~scope_exit()
        {
                if (execute_on_destruction_flag && execute_on_destruction())
                                exit_function();
        }
        void release() noexcept { execute_on_destruction_flag=false;}

        scope_exit(scope_exit const &)=delete;
        scope_exit& operator=(scope_exit const &)=delete;
        scope_exit& operator=(scope_exit &&)=delete;
private:
        EF  exit_function; // exposition only
        bool execute_on_destruction_flag{true}; // exposition only
        _detail::exception_counter ec; // exposition only
        bool execute_on_destruction()const { // exposition only
                switch(scope){ // not really a switch, since compile time constant
                case _detail::scope_run::fail:
                        return  ec.is_new_exception(); // only run if new exepc-
tion
                case _detail::scope_run::success:
                        return !ec.is_new_exception(); // only run if everything worked
fine
                case _detail::scope_run::exit:;
                }
                return true; // run always
        }
};

template <typename EF>
auto make_scope_exit(EF &&exit_function) noexcept {
        return
        scope_exit<
                std::conditional_t<
                        std::is_lvalue_reference<EF>{} && !std::is_const<EF>{},
                        EF, /* this is an lvalue reference */
                        std::remove_reference_t<EF>> /* store it by value, move/copy must not thro
                >(std::forward<EF>(exit_function));
}
```

```
template <typename EF>
auto make_scope_fail(EF &&exit_function) noexcept {
        return
        scope_exit<
                std::conditional_t<
                        std::is_lvalue_reference<EF>{} && !std::is_const<EF>{},
                        EF, /* this is an lvalue reference */
                        std::remove_reference_t<EF>> /* store it by value, move/copy must not throw */
                ,_detail::scope_run::fail
                >(std::forward<EF>(exit_function));
}

template <typename EF>
auto make_scope_success(EF &&exit_function) noexcept {
        return
        scope_exit<std::conditional_t<
                std::is_lvalue_reference<EF>{} && !std::is_const<EF>{},
                EF, /* this is an lvalue  reference */
                std::remove_reference_t<EF>> /* store it by value, move/copy must not throw */
                ,_detail::scope_run::success
        >(std::forward<EF>(exit_function));
}

}
}

#endif /* SCOPE_EXIT_H_ */
```

## 7.2 Unique Resource

```
#ifndef UNIQUE_RESOURCE_H_
#define UNIQUE_RESOURCE_H_
#include <type_traits>
namespace std{
namespace experimental{
namespace __detail {
template <typename D, typename R,typename=void>
struct provide_operator_arrow_for_pointer_to_class_types{}; // R is non-pointer or
pointer-to-non-class-type

template <typename DERIVED, typename R>
struct provide_operator_arrow_for_pointer_to_class_types<DERIVED, R,
        typename std::enable_if<std::is_pointer<R>::value
                && (
                                std::is_class<std::remove_pointer_t<R>>::value ||
                                std::is_union<std::remove_pointer_t<R>>::value )
                >::type >

{
```

```
        R operator->() const {
                return static_cast<const DERIVED*>(this)->get();
        }
};

}


template<typename R,typename D>
class unique_resource
                :public
                 __detail::provide_operator_arrow_for_pointer_to_class_types<
                         unique_resource<R,D>,R> {
        R resource; // exposition only
        D deleter;  // exposition only
        bool execute_on_destruction; // exposition only
public:
        // construction
        explicit
        unique_resource(R && resource,
                        std::conditional_t<std::is_reference<D>{},D,std::add_lvalue_referenc
                        bool shouldrun=true) noexcept
                :  resource{std::move(resource)}
                ,  deleter{deleter}
                , execute_on_destruction{shouldrun}{
                        static_assert(std::is_nothrow_move_constructible<R>{},"resource must
                }

        explicit
        unique_resource(R && resource,
                        std::remove_reference_t<D> && deleter, bool shouldrun=true) noexcept
                        :  resource{std::move(resource)}
                        ,  deleter{std::move(deleter)}
                        , execute_on_destruction{shouldrun}{
// static_assert(std::is_nothrow_move_constructible¡R¿,"resource must be nothrow_move_con-
structible");
                                static_assert(!std::is_lvalue_reference<D>{},"deleter must n
                        }

        // 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(unique_resource const &)=delete; // no copies!
        unique_resource& operator=(unique_resource  &&other)
```

```
                noexcept(noexcept(unique_resource::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;
        }
        unique_resource& operator=(unique_resource const &)=delete;
    // resource release
        ~unique_resource() //noexcept(noexcept(this-¿reset()))
        {
                this->reset();
        }
        void reset() //noexcept(noexcept(unique_resource::get_deleter()(resource)))
        {
                if (execute_on_destruction) {
                        this->execute_on_destruction = false;
                        this->get_deleter()(resource);
                }
        }
        void reset(R && newresource) //noexcept(noexcept(unique_resource::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 resource;
        }
        operator  R const &() const noexcept {
                return 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<remove_reference_t<R>,
                std::conditional_t<
                        std::is_lvalue_reference<D>{} && !std::is_const<D>{},
                        D, /* this is an lvalue reference */
                        std::remove_reference_t<D>> /* store it by value, move/copy must not throw
                >(std::move(r)
                ,std::forward<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
                ,std::conditional_t<
                        std::is_lvalue_reference<D>{} && !std::is_const<D>{},
                        D, /* this is an lvalue reference */
                        std::remove_reference_t<D>> /* store it by value, move/copy must not throw
                        >(std::move(r), std::forward<D>(d), shouldrun);
}



}
}
// example only, not specified.
#include <functional>
template <typename R>
auto
make_unique_resource_type_erased(R &&r
, std::function<void(std::remove_reference_t<R>)> &&d)
{
        return std::experimental::make_unique_resource(std::move(r),std::move(d));
}

#endif /* UNIQUE_RESOURCE_H_ */
```