d0052r4 - Generic Scope Guard and RAII Wrapper for the Standard Library

Peter Sommerlad and Andrew L. Sandoval
with contributions by Eric Niebler and Daniel Krügler

2017-02-17

Document Number: d0052r4	(update of N4189, N3949, N3830, N3677)
Date:	2017-02-17
Project:	Programming Language C++
Audience:	LWG/LEWG

1 History

1.1 Changes from P0052R3

— Use lowercase p and r like most other papers

— Change example code to use C++17 constructor template argument deduction (Yes, I checked my code with GCC, found "bugs" in the standard and change my/Eric's implementation to actually work with it).

1.2 Changes from P0052R2

— Take into account class template ctor argument deduction. However, I recommend keeping the factories for LFTS 3 to allow for C++14 implementations. At the time of this writing, I do not have a working C++17 compliant compiler handy to run corresponding test cases without the factories. However, there is one factory function make_unique_checked that needs to stay, because it addresses a specific but seemingly common use-case.

— Since `scope_success` is a standard library class that has a possible throwing destructor section [res.on.exception.handling] must be adjusted accordingly.

— The lack of factories for the classes might require explicit deduction guides, but I need help to specify those accordingly since I do not have a working C++17 compiler right at hand to test it.

1.3 Changes from P0052R1

The Jacksonville LEWG, especially Eric Niebler gave splendid input in how to improve the classes in this paper. I (Peter) follow Eric's design in specifying scope_exit as well as unique_resource in a more general way.

— Provide `scope_fail` and `scope_success` as classes. However, we may even hide all of the scope guard types and just provide the factories.

— safe guard all classes against construction errors, i.e., failing to copy the deleter/exit-function, by calling the passed argument in the case of an exception, except for scope_success.

— relax the requirements for the template arguments.

Special thanks go to Eric Niebler for providing several incarnations of an implementation that removed previous restrictions on template arguments in an exception-safe way (Eric: *"This is HARD."*). To cite Eric again: *"Great care must be taken when move-constructing or move-assigning unique_resource objects to ensure that there is always exactly one object that owns the resource and is in a valid, Destructible state."* Also thanks to Axel Naumann for presenting in Jacksonville and to Axel, Eric, and Daniel Krügler for their terrific work on wording improvements.

1.4 Changes from P0052R0

In Kona LWG gave a lot of feedback and especially expressed the desire to simplify the constructors and specification by only allowing *nothrow-copyable* `RESOURCE` and `DELETER` types. If a reference is required, because they aren't, users are encouraged to pass a `std::ref/std::cref` wrapper to the factory function instead.

— Simplified constructor specifications by restricting on nothrow copyable types. Facility is intended for simple types anyway. It also avoids the problem of using a type-erased `std::function` object as the deleter, because it could throw on copy.

— Add some motivation again, to ease review and provide reason for specific API issues.

— Make "Alexandrescu's" "declarative" scope exit variation employing `uncaught_exceptions()` counter optional factories to chose or not.

— propose to make it available for standalone implementations and add the header `<scope>` to corresponding tables.

— editorial adjustments

— re-established `operator*` for `unique_resource`.

— overload of `make_unique_resource` to handle `reference_wrapper` for resources. No overload for reference-wrapped deleter functions is required, because `reference_wrapper` provides the call forwarding.

1.5 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.

1.6 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.

1.7 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.

1.8 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 (resource acquisition is initialization) 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 Fall 2015(Kona),Fall 2014(Urbana), 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

While `std::unique_ptr` can be (mis-)used to keep track of general handle types with a user-specified deleter it can become tedious and error prone. Further argumentation can be found in previous papers. Here are two examples using `<cstdio>`'s `FILE *` and POSIX`<fcntl.h>`'s and `<unistd.h>`'s `int` file handles.

```
void demonstrate_unique_resource_with_stdio() {
        const std::string filename = "hello.txt";
        {
                unique_resource file(::fopen(filename.c_str(), "w"), &::fclose);
                ::fputs("Hello World!\n", file.get());
                ASSERT(file.get()!= NULL);
        }
        {
                std::ifstream input { filename };
                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*)
                ASSERT_EQUAL((FILE*)NULL, file.get());
        }
}
void demontrate_unique_resource_with_POSIX_IO() {
        const std::string filename = "./hello2.txt";
        {
                unique_resource file(::open(filename.c_str(), O_CREAT | O_RDWR | O_TRUNC, 0666), &::cl

                ::write(file.get(), "Hello World2!\n", 13u);
                ASSERT(file.get() != -1);
        }
        {
                std::ifstream input { filename };
                std::string line { };
                getline(input, line);
                ASSERT_EQUAL("Hello World2!", line);
                getline(input, line);
                ASSERT(input.eof());
        }
        ::unlink(filename.c_str());
        {
                auto file = make_unique_resource_checked(::open("nonexistingfile.txt", O_RDONLY), -1,
                ASSERT_EQUAL(-1, file.get());
        }
}
```

We refer to Andrej Alexandrescu's well-known many presentations as a motivation for `scope_exit`, `scope_fail`, and `scope_success`. Here is a brief example on how to use the 3 proposed factories.

```
void demo_scope_exit_fail_success(){
        std::ostringstream out{};
        auto lam=[&]{out << "called ";};
        try{
                scope_exit v([&]{out << "always ";});
                scope_success w([&]{out << "not ";}); // not called
                scope_fail x(lam); // called
                throw 42;
        }catch(...){
                scope_fail y([&]{out << "not ";}); // not called
                scope_success z([&]{out << "handled";}); // called
        }
        ASSERT_EQUAL("called always handled",out.str());
}
```

5 Impact on the Standard

This proposal is a pure library extension. A new header, `<scope>` is proposed, but it does not require changes to any standard classes or functions. Since it proposes a new header, no feature test macro seems required. It does not require any changes in the core language, and it has been implemented in standard C++ conforming to C++14, resp. draft C++17. Depending on the timing of the acceptance of this proposal, it might go into a library fundamentals TS under the namespace std::experimental or directly in the working paper of the standard.

6 Design Decisions

6.1 General Principles

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

— 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` and `scope_exit`/`scope_fail` are needed. Therefore, the specification asks for strong safety guarantee when creating and moving the defined types, making sure to call the deleter/exit function if such attempts fail.

— 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.

— Should we make the regular constructor of the scope guard templates private and friend the factory function only? This could prohibit the use as class members, which might sneakily be used to create "destructor" functionality by not writing a destructor by adding a `scope_exit` member variable.
It seems C++17's class template constructor argument deduction makes the need for most of the factory functions obsolete and thus this question is no longer relevant. However, I recommend keeping the factories for the LFTS-3 if accepted to allow backporting to C++14.

— Should the scope guard classes be move-assignable? Doing so, would enable/ease using them as class members. I do not think this use is good, but may be someone can come up with a use case for that.
LEWG already answered that once with NO, but you never know if people change their mind again.

The following issues have been recommended by LWG already:

— Make it a facility available for free-standing implementations in a new header `<scope>` (`<utility>` doesn't work, because it is not available for free-standing implementations)

6.3 Open Issues (to be) Discussed by LEWG / LWG

Since a lot of the design changed due to Eric Niebler's implementation taking great effort to provide strong exception guarantee, another round through LEWG and LWG seems to be required.

— How do we specify deduction guides in the standard? Synopsis only? Should they be `explicit` (My gcc version doesn't allow that syntax yet)?

— which "callable" definition in the standard should be applied (call expression (as it is now) or via INVOKE (`is_callable_v<EF&>`). *IMHO call expression is fine, since everything is about side-effects and we never return a useful value from any of the function objects.*

— Should we provide a non-explicit conversion operator to `R` in `unique_resource<R,D>` ? Last time people seem to have been strongly against, however, it would make the use of unique_resource much easier in contexts envisioned by author Andrew Sandoval. Please re-visit, since it is omitted here.

— It would be nice to get feedback on my adapted implementation, since I am not sure, my deduction guides will actually behave identical to the factory functions provided by Eric.

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::`.

A very recent draft of the standard already has the requested change below that was suggested by Daniel Krügler:

7.1 Adjust 17.5.4.8 Other functions [res.on.functions]

Since `scope_success()` might throw an exception and we can not specify that in a required behavior clause, we need to allow doing so for the standard library's normative remarks section as well.

In section 17.5.4.8 Other functions [res.on.functions] modify p2 item (2.4) as follows by adding "or *Remarks:* "

(2.4) — if any replacement function or handler function or destructor operation exits via an exception, unless specifically allowed in the applicable *Required behavior:* **or *Remarks:*** paragraph.

However the following adjustment is missing, since the standard library promises that all library classes won't throw on destruction:

7.2 Adjust 17.5.5.12 Restrictions on exception handling [res.on.exception.handling]

Change paragraph 3 as follows:

1 Destructor operations defined in the C++ standard library shall not throw exceptions unless explicitly specified that they could. Every destructor without an exception specification in the C++ standard library shall behave as if it had a non-throwing exception specification.

7.3 Header

In section 17.5.1.1 Library contents [contents] add an entry to table 14 for the new header `<scope>`. Because of the new header, there is no need for a feature test macro.

In section 17.5.1.3 Freestanding implementations [compliance] add an extra row to table 16 (cpp.library.headers) and in section [utilities.general] add the same extra row to table 44 (util.lib.summary)

Table 1 — table 16 and table 44

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

7.4 Additional sections

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

7.5 Scope guard support [scope]

This subclause contains infrastructure for a generic scope guard and RAII (resource acquisition is initialization) resource wrapper.

7.5.1 Header <scope> synopsis

```
namespace std {
namespace experimental {
template <class EF>
class scope_exit;
template <class EF>
class scope_fail;
template <class EF>
class scope_success;


template<class R,class D>
class unique_resource;

// special factory function
template<class R,class D, class S=R>
unique_resource<decay_t<R>, decay_t<D>>
make_unique_resource_checked(R&& r, S const& invalid, D&& d)
noexcept(is_nothrow_constructible_v<decay_t<R>, R> &&
         is_nothrow_constructible_v<decay_t<D>, D>);

// deduction guides
template <class EF>
scope_exit(EF &&ef) -> scope_exit<std::decay_t<EF>>;
template <class EF>
scope_fail(EF &&ef) -> scope_fail<std::decay_t<EF>>;
template <class EF>
scope_success(EF &&ef) -> scope_success<std::decay_t<EF>>;

template<typename R, typename D>
unique_resource(R &&r, D &&d) -> unique_resource<std::decay_t<R>, std::decay_t<D>>;




// optional factory functions (should at least be present for LFTS3)
template <class EF>
scope_exit<decay_t<EF>> make_scope_exit(EF&& exit_function) ;
template <class EF>
scope_fail<decay_t<EF>> make_scope_fail(EF&& exit_function) ;
template <class EF>
scope_success<decay_t<EF>> make_scope_success(EF&& exit_function) ;
```

```
template<class R,class D>
unique_resource<decay_t<R>, decay_t<D>>
make_unique_resource( R&&  r, D&& d)
noexcept(is_nothrow_constructible_v<decay_t<R>, R> &&
         is_nothrow_constructible_v<decay_t<D>, D>);

template<class R,class D>
unique_resource<R&, decay_t<D>>
make_unique_resource( reference_wrapper<R>  r, D&& d)
noexcept(is_nothrow_constructible_v<decay_t<D>, D>);

}}
```

[1] The header `<scope>` defines the class templates `scope_exit`, `scope_fail`, `scope_success`, `unique_resource` and the factory function template `make_unique_resource_checked()`. [*Note:* For backporting possibilities I recommend that the following factory functions are kept, even they are made obsolete by C++17's constructor template argument deduction: `make_scope_exit()`, `make_scope_success()`, `make_scope_fail()`, and `make_unique_resource()`. — *end note*]

[2] The class templates `scope_exit`, `scope_fail`, and `scope_success` define *scope guards* that wrap a function object to be called on their destruction.

[3] The following clauses describe the class templates `scope_exit`, `scope_fail`, and `scope_success`. In each clause, the name *`scope_guard`* denotes either of these class templates. In description of class members *`scope_guard`* refers to the enclosing class.

7.5.2 Scope guard class templates [scope.scope_guard]

```
template <class EF>
class scope_guard {
public:
  template<class EFP>
  explicit scope_guard(EFP&& f) ;
  scope_guard(scope_guard&& rhs) ;
  ~scope_guard() noexcept(see below);
  void release() noexcept;

  scope_guard(const scope_guard&)=delete;
  scope_guard& operator=(const scope_guard&)=delete;
  scope_guard& operator=(scope_guard&&)=delete;
private:
  EF exit_function;     // exposition only
  bool execute_on_destruction{true}; //exposition only
  int  uncaught_on_creation{uncaught_exceptions()}; // exposition only
};
```

1 [*Note:* `scope_exit` is meant to be a general-purpose scope guard that calls its exit function when a scope is exited. The class templates `scope_fail` and `scope_success` share the `scope_exit`'s interface, only the situation when the exit function is called differs. These latter two class templates memorize the value of `uncaught_exceptions()` on construction and in the case of `scope_fail` call the exit function on destruction, when `uncaught_exceptions()` at that time returns a greater value, in the case of `scope_success` when `uncaught_exceptions()` on destruction returns the same or a lesser value.
[*Example:*

```
void grow(vector<int>&v){
        scope_success guard([]{ cout << "Good!" << endl; });
        v.resize(1024);
} // outputs "Good!" unless resize() fails
```

— *end example*] — *end note*]

2 *Requires:* Template argument `EF` shall be a function object type ([function.objects]), lvalue reference to function, or lvalue reference to function object type. If EF is an object type, it shall satisfy the requirements of `Destructible` (Table 24). Given an lvalue `f` of type `EF`, the expression `f()` shall be well formed and shall have well-defined behavior. The constructor arguments `f` in the following constructors shall be a function object (20.9)[function.objects], lvalue reference to function, or lvalue reference to function object.

```
template<class EFP>
explicit
scope_exit(EFP&& f) ;
```

Remarks: This constructor shall not participate in overload resolution unless `is_constructible_v<EF,EFP>` is `true`.

3 *Requires:* Given an lvalue `f` of type `EFP`, the expression `f()` shall be well formed and shall have well-defined behavior.

4 *Effects:* If `EFP` is not a lvalue-reference type and `is_nothrow_constructible_v<EF,EFP>` is `true`, initialize `exit_function` with `move(f)` otherwise initialize `exit_function` with `f`. If the copying throws an exception, calls `f()`.

5 *Throws:* Any exception thrown during the copying of `f`.

```
template<class EFP>
explicit
scope_fail(EFP&& f) ;
```

6 *Remarks:* This constructor shall not participate in overload resolution unless `is_constructible_v<EF,EFP>` is `true`.

7 *Requires:* Given an lvalue `f` of type `EFP`, the expression `f()` shall be well formed and shall have well-defined behavior.

8 *Effects:* If `EFP` is not a lvalue-reference type and `is_nothrow_constructible_v<EF,EFP>` is `true`, initialize `exit_function` with `move(f)` otherwise initialize `exit_function` with `f`. If the copying throws an exception, calls `f()`.

9 *Throws:* Any exception thrown during the copying of `f`.

```
template<class EFP>
explicit
scope_success(EFP&& f) ;
```

10 *Remarks:* This constructor shall not participate in overload resolution unless `is_constructible_v<EF,EFP>` is `true`.

11 *Effects:* If `EFP` is not a lvalue-reference type and `is_nothrow_constructible_v<EF,EFP>` is `true`, initialize `exit_function` with `move(f)` otherwise initialize `exit_function` with `f`. [*Note:* If copying fails, `f()` won't be called. — *end note*]

12 *Throws:* Any exception thrown during the copying of `f`.

`scope_guard``(`*`scope_guard`*`&& rhs) ;`

13 *Effects:* `execute_on_destruction` yields the value `rhs.execute_on_destruction` yielded before the construction. If `is_nothrow_move_constructible_v<EF>` move constructs otherwise copy constructs `exit_function` from `rhs.exit_function`. If construction succeeds, call `rhs.release()`.

14 *Throws:* Any exception thrown during the copying of `rhs.exit_function`.

```
~scope_exit() noexcept(true);
```

15 *Effects:*

```
if (execute_on_destruction)
        exit_function();
```

```
~scope_fail() noexcept(true);
```

16 *Effects:*

```
if (execute_on_destruction
   && uncaught_exceptions() > uncaught_on_creation)
```

```
            exit_function();
```

```
~scope_success() noexcept(noexcept(exit_function()));
```

17 *Effects:*

```
  if (execute_on_destruction
     && uncaught_exceptions() <= uncaught_on_creation)
          exit_function();
```

18 *Remarks:* If `noexcept(exit_function())` is `false`, `exit_function()` may throw an exception, notwithstanding the restrictions of [res.on.exception.handling].

19 *Throws:* If `noexcept(exit_function())` is `false`, throws any exception thrown by `exit_function()`.

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

20 `execute_on_destruction=false;`

7.5.3 Scope guard factory functions [scope.make_scope_exit]

[*Note:* Should I specify the deduction guides here. Or should they go into the synopsis (only)? — *end note*] [*Note:* These factory functions are meant for C++14 backward compatibility of this feature. — *end note*]

1 The scope guard factory functions create `scope_exit`, `scope_fail`, and `scope_success` objects that for the function object `exit_function` evaluate `exit_function()` at their destruction unless `release()` was called.

```
template <class EF>
scope_exit<decay_t<EF>> make_scope_exit(EF&& exit_function) ;
```

2 *Returns:* `scope_exit<decay_t<EF»(forward<EF>(exit_function));`

```
template <class EF>
scope_fail<decay_t<EF>> make_scope_fail(EF&& exit_function) ;
```

3 *Returns:* `scope_fail<decay_t<EF»(forward<EF>(exit_function));`

```
template <class EF>
scope_success<decay_t<EF>> make_scope_success(EF&& exit_function) ;
```

6 *Returns:* `scope_success<decay_t<EF»(forward<EF>(exit_function));`

7.5.4 Unique resource wrapper [scope.unique_resource]

7.5.5 Class template `unique_resource` [scope.unique_resource.class]

```
template<class R,class D>
class unique_resource {
public:
  template<class RR, class DD>
  explicit unique_resource(RR&& r, DD&& d)
    noexcept(is_nothrow_constructible_v<R, RR>&&
             is_nothrow_constructible_v<D, DD>)
  unique_resource(unique_resource&& rhs)
    noexcept(is_nothrow_move_constructible_v<R> &&
             is_nothrow_move_constructible_v<D>);
  unique_resource(const unique_resource&)=delete;
  ~unique_resource();
  unique_resource& operator=(unique_resource&& rhs) ;
  unique_resource& operator=(const unique_resource&)=delete;
  void swap(unique_resource& other);
  void reset();
  template<class RR>
  void reset(RR&& r);
  void release() noexcept;
  const R& get() const noexcept;
  R operator->() const noexcept;
  see below operator*() const noexcept;
  const D& get_deleter() const noexcept;
private:
  R resource; // exposition only
  D deleter; // exposition only
  bool execute_on_destruction{true}; // exposition only
};
```

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 `get()` and in case of a pointer type resource through a set of convenience pointer operator functions. — *end note*]

2 The template argument `D` shall be a Destructible (Table 24) function object type (20.9), for which, given a value `d` of type `D` and a value `r` of type `R`, the expression `d(r)` shall be well formed, shall have well-defined behavior, and shall not throw an exception.

3 `R` shall be a Destructible (Table 24) object type or a lvalue reference type.

4 [*Note:* In case of `R` being a lvalue reference, an implementation should chose `reference_wrapper<R>` as type for `resource` and adjust functionality accordingly by delegating to `resource.get()` in appropriate places. In case of `D` being a `reference_wrapper` instantiation no special delegation is required, since `reference_wrapper` already delegates the call operator ([refwrap.invoke]). — *end note*]

5 *Requires:* (is_copy_constructible_v<R> || is_nothrow_move_constructible_v<R>) && (is_copy_constructible_v<D> || is_nothrow_move_constructible_v<D>)

```
template<class RR, class DD>
explicit unique_resource(RR&& r, DD&& d)
    noexcept(is_nothrow_constructible_v<R, RR>
            && is_nothrow_constructible_v<D, DD>)
```

6 *Remarks:* given

```
template<class T, class TT>
using is_ntmocp_constructible =
     conditional_t<
        is_reference_v<TT> || !is_nothrow_move_constructible_v<TT>,
        is_constructible<T, TT const &>,
        is_constructible<T, TT>>;
template<class T, class TT>
constexpr auto is_nothrow_move_or_copy_constructible_from_v=
                is_ntmocp_constructible<T,TT>::value;
```

this constructor shall not participate in overload resolution unless *is_nothrow_move_or_copy_constructible_from_v*<R, RR> is true and *is_nothrow_move_or_copy_constructible_from_v*<D, DD> is true.

7 *Effects:* If RR is not a lvalue-reference and is_nothrow_constructible<R,RR> is true, initializes resource with move(r), otherwise initializes resource with r. Then, if DD is not an lvalue reference and is_nothrow_constructible<D,DD> is true, initializes deleter with move(d), otherwise initializes deleter with d. If construction of resource throws an exception, calls d(r). If construction of deleter throws an exception, calls d(resource). [*Note:* The explained mechanism should ensure no leaking resources. — *end note*]

8 *Throws:* any exception thrown during construction.

```
unique_resource(unique_resource&& rhs)
  noexcept(is_nothrow_move_constructible_v<R> &&
           is_nothrow_move_constructible_v<D>);
```

9 *Effects:* If is_nothrow_move_constructible_v<R> is true, initialize resource from forward<R>(rhs.resour otherwise initialize resource from rhs.resource. [*Note:* If construction of resource throws an exception rhs is left owning the resource and will free it in due time. — *end note*] Then if is_nothrow_move_constructible_v<D> is true initialize deleter from forward<D>(rhs.deleter), otherwise initialize deleter from rhs.deleter. If construction of deleter throws an exception: if !is_nothrow_move_constructible_v<R>, then rhs.deleter(resource) ; rhs.release(); otherwise rhs.resource and rhs.deleter are unmodified and rhs can be left owning the resource. Finally, execute_on_destruction is initialized with exchange(rhs.execute_on_-destruction,false). [*Note:* The explained mechanism should ensure no leaking resources. — *end note*]

```
unique_resource& operator=(unique_resource&& rhs) ;
```

10 *Requires:*

`(is_nothrow_move_assignable_v<R> || is_copy_assignable_v<R>)` and
`(is_nothrow_move_assignable_v<D> || is_copy_assignable_v<D>)`

11 *Effects:* If `this == &rhs` no effect. Otherwise `reset();` followed by Given the members `resource` and `deleter` If `nothrow_move_assignable_v<R>`, try to copy or move assign `deleter` from `rhs.deleter` first then `resource=forward<R>(rhs.resource)`, if `nothrow_move_assignable_v<D>`, try to copy or move `resource` from `rhs.resource` first then `deleter=forward<D>(rhs.deleter)`, otherwise try to copy the two members. Then `execute_on_destruction = exchange(rhs.execute_on_destruction, false);`. [*Note:* If a copy of a member throws an exception this mechanism leaves `rhs` intact and `*this` in the released state. — *end note*]

12 *Throws:* Any exception thrown during a copy-assignment of a member that can not be moved without an exception.

13 [*Note:* The move semantics differ from `pair` [pairs.pair] and `tuple`[tuple.tuple], because it is important that `unique_resource` will not leak its held resource. — *end note*] .

```
~unique_resource();
```

14 *Effects:* `reset()`.

```
void reset();
```

15 *Effects:* Equivalent to

```
if (execute_on_destruction) {
  execute_on_destruction=false;
  get_deleter()(resource);
}
```

```
template <class RR>
void reset(RR && r) ;
```

16 Given

```
template<class T>
constexpr conditional_t<
    (!is_nothrow_move_assignable_v<T> &&
      is_copy_assignable_v<T>),
    T const &,
    T &&>
move_assign_if_noexcept(T &x) noexcept
{
    return std::move(x);
}
```

17 *Remarks:* This function shall not participate in overload resolution if `resource = move_assign_if_noexcept(r)` is ill-formed.

18 *Effects:* Equivalent to

```
reset();
resource = move_assign_if_noexcept(r);
execute_on_destruction = true;
```

If copy-assignment of `resource` throws an exception, `get_deleter()(r)`.

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

19 *Effects:* `execute_on_destruction = false`.

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

20 *Remarks:* Member `operator->` shall not participate in overload resolution unless `is_pointer_v<R> && is_nothrow_copy_constructible_v<R> &&(is_class_v<remove_pointer_t<R» || is_union_v<remove_pointer_t<R»)` is `true`.

21 *Returns:* `resource`.

```
see below operator*() const noexcept ;
```

22 *Remarks:* Member `operator*` shall not participate in overload resolution unless `is_pointer_v<R>` is `true`.

23 *Returns:* `*resource`. The return type is equivalent to `add_lvalue_reference_t<remove_pointer_t<R»`.

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

24 *Returns:* `deleter`

7.5.6 Factories for `unique_resource` [scope.make_unique_resource]

[*Note:* The make_unique_resource factory functions can be omitted for putting it into the standard. However, they could be useful for the LFTS3 to allow backporting to C++14. However, omitting these factories might require adaptation of the constructors to get template argument deduction correct. — *end note*]

```
template<class R,class D, class S=R>
unique_resource<decay_t<R>,decay_t<D>>
make_unique_resource_checked(R&& r, S const & invalid, D && d )
noexcept(is_nothrow_constructible_v<decay_t<R>, R> &&
         is_nothrow_constructible_v<decay_t<D>, D>);
```

1 *Requires:* If `s` denotes a (possibly const) value of type `S` and `r` denotes a (possibly const) value of type `R`, the expressions `s == r` and `r == s` are both valid, both have the same domain, both have a type that is convertible to `bool`, and `bool(s == r) == bool(r == s)` for every `r` and `s`. If `S` is the same type as `R`, `R` shall be EqualityComparable(Table 17).

2 *Effects:* As if

```
bool mustrelease = bool(r == invalid);
auto ur= unique_resource(forward<R>(r), forward<D>(d));
if(mustrelease) ur.release();
return ur;
```

3 [*Note:* This factory function exists to avoid calling a deleter function with an invalid argument. The following example shows its use to avoid calling `fclose` when `fopen` failed and returned `NULL`. — *end note*]

4 [*Example:*

```
auto file = make_unique_resource_checked(
      ::fopen("potentially_nonexisting_file.txt", "r"),
      (FILE*) NULL, &::fclose);
```

— *end example*]

[*Note:* The following functions are optional and could be omitted for C++next. — *end note*]

```
template<class R,class D>
unique_resource<decay_t<R>, decay_t<D>>
make_unique_resource( R && r, D && d)
noexcept(is_nothrow_constructible_v<decay_t<R>, R> &&
         is_nothrow_constructible_v<decay_t<D>, D>);
```

5 *Returns:*
`unique_resource<decay_t<R>, decay_t<D»(forward<R>(r), forward<D>(d))`

```
template<class R,class D>
unique_resource<R&,decay_t<D>>
make_unique_resource( reference_wrapper<R> r, D d)
noexcept(is_nothrow_constructible_v<decay_t<D>, D>);
```

6 *Returns:* `unique_resource<R&,decay_t<D»(r.get(),forward<D>(d))`

7 [*Note:* There is no need to overload on `reference_wrapper<D>` for the deleter. — *end note*]

8 Appendix: Example Implementations

removed, see
https://github.com/PeterSommerlad/SC22WG21_Papers/tree/master/workspace/P0052_scope_-exit/src