p0052r7 - Generic Scope Guard and RAII Wrapper for the Standard Library

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

2018-03-16

Document Number: p0052r7	(update of N4189, N3949, N3830, N3677)
Date:	2018-03-16
Project:	Programming Language C++
Audience:	LWG

1 History

1.1 Changes from P0052R6

— remove the phrasing for potentially targeting a TS and minor editorial fixes.

— rephrased p3 in [scope.unique_resource.class] according to Alisdair Meredith's suggestion.

— corrected code in specification of `unique_resource` move assignment operator to use `R1` instead of R, since R1 is the type of the member `resource`, and replace `std::forward()` with `std::move()` and to rely only on `std::is_nothrow_move_assignable_v` in noexcept specification explanation and `if constexpr` condition.

— add "calls" to deleter() in reset()

— added requirement to `unique_resource` constructor to clarify that calling `d()` is always possible and fixed effects to unwrap `resource`, if it is stored in a `reference_wrapper` (thanks Tim Song).

— changed `std::forward<D>(rhs.deleter)` to `std::move(rhs.deleter)` in the effects of the move-constructor of `unique_resource`, because `D` can not be a reference type. (thanks Tim Song). Can not do that for `resource`, because the member variable might be a `reference_wrapper`.

— simplify `get()` because `reference_wrapper` auto-converts to const reference.

— adjust title of [scope.make_unique_resource]

— added requires clause to `unique_resource::reset(RR)` to allow for running the deleter on the function argument in case of an exception (thanks Tim Song).

— default argument for template parameter `S` in `make_unique_resource_checked` is now `decay_t<R>` (thanks Tim Song).

— comparison in `make_unique_resource_checked` is now required to not throw an exception.

— In Jacksonville LWG was discussion the need for `decay_t<R>` in `make_unique_resource_checked` and asked for an lvalue version of a call in the example. That is not what it should be used for anyway. I made some experiments and made the decision that was already previously made (if I remember correctly) to only support copying of the resource in that factory function. Therefore, `decay_t` is IMHO the right thing to do. and therefore I also did not add an example using an lvalue for its first argument.

1.2 Changes from P0052R5

Wording reviewed and recommended on by LWG

— added noexcept specification for move assignment.

— feature test macro added `__cpp_lib_scope`

— drop `unique_resource` deduction guide that unwraps reference_wrapper

— add a non-normative note to explain potential scope guard misuse if capturing local by reference that is returned. BSI raised this issue, but does not intend to ask this paper to solve that corner case.

— the code in the special factory function's effects was broken, but can be fixed in an implementation. Changed the specification into words, so that implementers can do the right thing. Note, previous versions of the paper had a specification with an extra bool constructor parameter to unique_resource achieving that mechanism.

— fixed some minor editorial things and forgotten changes

— separated definitions of `unique_resource` member functions returning the resource.

— simplified specification of `reset` using `if constexpr` according to Jonathan Wakely without inventing an exposition only function. (this must be re-checked)

— more fancy attempt to specify the need for implementations to internally use `reference_wrapper` in `unique_resource` if the resource type is a reference (to support assignment) by specifying a separate type in the `unique_resource` synopsis for resource and clarifying the note saying to use reference_wrapper.

— removed remains of `swap()` that got not deleted.

— simplified `unique_resource` specification as suggested by Stephan T. Lavavej

1.3 Changes from P0052R4

Wording reviewed and recommended on by LWG

— Add missing deduction guides

— Call expressions are OK.

— No consensus to re-add the implicit conversion operator to unique_resource

— clarification of wording in many places

1.4 Changes from P0052R3

— Take new section numbering of the standard working paper into account.

— require noexcept of `f()` for `scope_exit` and `scope_fail` explicitly

— implementation could be tested with C++17 compiler and class template constructor argument deduction thus the paper no longer claims help or not being sure.

1.5 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.

1.6 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.

1.7 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.

1.8 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.

1.9 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.

1.10 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.

1.11 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).

— Thanks to participants in LWG in Jacksonville, Toronto and Albuquerque, especially STL, Lisa Lippincott, Casey Carter, many others, and Marshall Clow for help with phrasing the wording.

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";
  { auto file=make_unique_resource(::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*) 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.get(), "Hello World!\n", 12u);
    ASSERT(file.get() != -1);
  }
  { 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(::open("nonexistingfile.txt",
                      O_RDONLY), -1, &::close);
    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{
    auto v=make_scope_exit([&]{out << "always ";});
    auto w=make_scope_success([&]{out << "not ";}); // not called
    auto x=make_scope_fail(lam); // called
    throw 42;
  }catch(...){
    auto y=make_scope_fail([&]{out << "not ";}); // not called
    auto z=make_scope_success([&]{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++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. I suggest both shipping vehicles.

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

The following issues have been resolved finally by LWG in Toronto. The shipping vehicle should be a new version of the Library Fundamentals TS, however, I would not object to put it directly into C++20.

- ~~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.~~

7 Technical Specifications

The following formulation is based on inclusion to the draft of the C++ standard.

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

7.1 Adjust 20.5.4.8 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 20.5.4.8 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 20.5.5.12 Restrictions on exception handling [res.on.exception.handling]

Change paragraph 3 as follows:

1 Unless otherwise specified, d~~D~~estructor operations defined in the C++ standard library shall not throw exceptions. 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 20.5.1.1 Library contents [contents] add an entry to table 16 (cpp.library.headers) for the new header `<scope>`.

In section 20.5.1.3 Freestanding implementations [compliance] add an extra row to table 19 (cpp.headers.freestanding) and in section [utilities.general] add the same extra row to table 34 (util.lib.summary)

Table 1 — table 19 and table 34

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

7.4 Additional sections

Add a a new section to chapter 23 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 [scope.syn]

```
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, const S& invalid, D&& d) noexcept(see below);
}}
```

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().

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 sections describe the class templates scope_exit, scope_fail, and scope_success. In each section, the name *scope_guard* denotes any of these class templates. In descriptions of the 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)
  noexcept(is_nothrow_constructible_v<EF, EFP>);
  scope_guard(scope_guard&& rhs) noexcept(see below);
  ~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
};

template <class EF>
scope_guard(EF) -> scope_guard<EF>;
```

1 `scope_exit` is 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` interface, only the situation when the exit function is called differs.
[*Example:*

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

— *end example*]

2 [*Note:* If the exit function object of a `scope_success` or `scope_exit` object refers to a local variable of the function where it is defined, e.g., as a lambda capturing the variable by reference, and that variable is used as a return operand in that function, that variable might have already been returned when the *`scope_guard`*'s destructor executes, calling the exit function. This can lead to surprising behavior. — *end note*]

3 *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 27). Given an lvalue `g` of type `remove_reference_t<EF>`, the expression `g()` shall be well-formed and shall have well-defined behavior.

4 The constructor argument in the following constructors shall be a function object([function.objects]), lvalue reference to function, or lvalue reference to function object.

```
template <class EFP>
explicit
scope_exit(EFP&& f) noexcept(is_nothrow_constructible_v<EF, EFP>);
```

5 *Remarks:* This constructor shall not participate in overload resolution unless `is_same_v<remove_cvref_t<EFP>, scope_exit<EF>>` is `false` and `is_constructible_v<EF, EFP>` is `true`.

6 *Requires:* Given an lvalue `f` of type `remove_reference_t<EFP>`, the expression `f()` shall be well-formed, have well-defined behavior, and not throw an exception.

7 *Effects:* If `EFP` is not an lvalue reference type and `is_nothrow_constructible_v<EF,EFP>` is `true`, initialize `exit_function` with `std::forward<EFP>(f)` otherwise initialize `exit_function` with `f`. If the initialization of `exit_function` throws an exception, calls `f()`.

8 *Throws:* Nothing, unless the initialization of `exit_function` throws.

```
template <class EFP>
explicit
scope_fail(EFP&& f) noexcept(is_nothrow_constructible_v<EF, EFP>);
```

9 *Remarks:* This constructor shall not participate in overload resolution unless `is_same_v<remove_cvref_t<EFP>, scope_fail<EF>>` is `false` and `is_constructible_v<EF,EFP>` is `true`.

10 *Requires:* Given an lvalue `f` of type `remove_reference_t<EFP>`, the expression `f()` shall be well-formed, have well-defined behavior, and not throw an exception.

11 *Effects:* If `EFP` is not an lvalue reference type and `is_nothrow_constructible_v<EF,EFP>` is `true`, initialize `exit_function` with `std::forward<EFP>(f)` otherwise initialize `exit_function` with `f`. If the initialization of `exit_function` throws an exception, calls `f()`.

12 *Throws:* Nothing, unless the initialization of `exit_function` throws.

```
template <class EFP>
explicit
scope_success(EFP&& f) noexcept(is_nothrow_constructible_v<EF, EFP>);
```

13 *Remarks:* This constructor shall not participate in overload resolution unless `is_same_v<remove_cvref_t<EFP>, scope_success<EF>>` is `false`, and `is_constructible_v<EF,EFP>` is `true`.

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

15 *Effects:* If `EFP` is not an lvalue reference type and `is_nothrow_constructible_v<EF,EFP>` is `true`, initialize `exit_function` with `std::forward<EFP>(f)` otherwise initialize `exit_function` with `f`. [*Note:* If initialization of `exit_function` fails, `f()` won't be called. — *end note*]

16 *Throws:* Nothing, unless the initialization of `exit_function` throws.

```
scope_guard(scope_guard&& rhs) noexcept(see below);
```

17 *Remarks:* The expression inside `noexcept` is equivalent to `is_nothrow_move_constructible_v<EF> || is_nothrow_copy_constructible_v<EF>`. This constructor does not take part in overload resolution unless the expression `(is_nothrow_move_constructible_v<EF> || is_copy_constructible_v<EF>)` is `true`.

18 *Effects:* If `is_nothrow_move_constructible_v<EF>` move constructs otherwise copy constructs `exit_function` from `rhs.exit_function`. If construction succeeds, call `rhs.release()`. [*Note:* Copying instead of moving provides the strong exception guarantee. — *end note*]

19 *Postconditions:* `execute_on_destruction` yields the value `rhs.execute_on_destruction` yielded before the construction. `uncaught_on_creation` yields the value `rhs.uncaught_on_creation` yielded before the construction.

20 *Throws:* Any exception thrown during the initialization of `exit_function`.

`~scope_exit() noexcept(true);`

21 *Effects:* Equivalent to:

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

`~scope_fail() noexcept(true);`

22 *Effects:* Equivalent to:

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

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

23 *Effects:* Equivalent to:

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

24 [*Note:* If `noexcept(exit_function())` is `false`, `exit_function()` may throw an exception, notwithstanding the restrictions of [res.on.exception.handling]. — *end note*]

25 *Throws:* Any exception thrown by `exit_function()`.

`void release() noexcept;`

26 *Effects:* Equivalent to `execute_on_destruction = false`.

7.5.3 Unique resource wrapper [scope.unique_resource]

7.5.4 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(see below);
  unique_resource(unique_resource&& rhs) noexcept(see below);
  ~unique_resource();
  unique_resource& operator=(unique_resource&& rhs) noexcept(see below);
  void reset() noexcept;
  template <class RR>
    void reset(RR&& r);
  void release() noexcept;
  const R& get() const noexcept;
  see below operator*() const noexcept;
  R operator->() const noexcept;
  const D& get_deleter() const noexcept;
private:
  using R1 = conditional_t< is_reference_v<R>, reference_wrapper<R>, R >; // exposition only
  R1   resource; // exposition only
  D    deleter;  // exposition only
  bool execute_on_destruction{true}; // exposition only
};

template<typename R, typename D>
  unique_resource(R, D)
    -> unique_resource<R, D>;
```

1 [*Note:* `unique_resource` is a universal RAII wrapper for resource handles. 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 27) function object type (23.14), 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. `D` shall either be CopyConstructible (Table 24), or `D` shall be MoveConstructible (Table 23) and `is_nothrow_move_constructible_v<D>` shall be `true`.

3 For the purpose of this sub-clause, a *resource type* `T` is an object type that is CopyConstructible (Table 24), or is an object type that is MoveConstructible (Table 23) and `is_nothrow_move_constructible_v<T>` is `true`, or is an lvalue reference to a resource type. `R` shall be a resource type.

7.5.5 `unique_resource` constructors [scope.unique_resource.ctor]

```
template <class RR, class DD>
explicit
unique_resource(RR&& r, DD&& d) noexcept(see below)
```

1 *Requires:* The expressions `d(r)` shall be well-formed, shall have well-defined behavior, and shall not throw an exception. If `is_reference_v<R>` is true, `d(resource.get())` otherwise `d(resource)` shall be well-formed, shall have well-defined behavior, and shall not throw an exception.

2 *Remarks:* The expression inside `noexcept` is equivalent to `is_nothrow_constructible_v<R, RR> && is_nothrow_constructible_v<D, DD>`. Given the following exposition only variable template

```
template <class T, class TT>
constexpr bool is_nothrow_move_or_copy_constructible_from_v =
    (is_reference_v<TT> || !is_nothrow_constructible_v<T,TT>)?
    is_constructible_v<T, TT const &>:
    is_constructible_v<T, TT>;
```

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`.

3 *Effects:* If `is_nothrow_constructible_v<R,RR>` is `true`, initializes `resource` with `std::forward<RR>(r)`, otherwise initializes `resource` with `r`. Then, if `is_nothrow_constructible_v<D,DD>` is `true`, initializes `deleter` with `std::forward<DD>(d)`, otherwise initializes `deleter` with `d`. If initialization of `resource` throws an exception, calls `d(r)`. If initialization of `deleter` throws an exception, if `is_reference_v<R>` is true, calls `d(resource.get())` otherwise calls `d(resource)`. [*Note:* The explained mechanism ensures no leaking of resources. — *end note*]

4 *Throws:* Any exception thrown during initialization.

```
unique_resource(unique_resource&& rhs) noexcept(see below)
```

5 *Remarks:* The expression inside `noexcept` is equivalent to `is_nothrow_move_constructible_v<R> && is_nothrow_move_constructible_v<D>`.

6 *Effects:* First, initialize `resource` as follows:

(6.1) — If `is_nothrow_move_constructible_v<R>` is `true`, from `std::forward<R>(rhs.resource)`,

(6.2) — otherwise, from `rhs.resource`.

7 [*Note:* If initialization of `resource` throws an exception, `rhs` is left owning the resource and will free it in due time. — *end note*]

8 Then, initialize `deleter` as follows:

(8.1) — If `is_nothrow_move_constructible_v<D>` is `true`, from `std::move(rhs.deleter)`;

(8.2) — otherwise, from `rhs.deleter`.

9 If initialization of `deleter` throws an exception and if `is_nothrow_move_constructible_v<R>`

is `true`:

```
rhs.deleter(resource); rhs.release();
```

Finally, `execute_on_destruction` is initialized with `exchange(rhs.execute_on_destruction,false)`.

[*Note:* The explained mechanism ensures no leaking of resources. — *end note*]

7.5.6 `unique_resource` assignment [scope.unique_resource.assign]

```
unique_resource& operator=(unique_resource&& rhs) noexcept(see below);
```

Remarks: The expression inside `noexcept` is equivalent to `is_nothrow_move_assignable_v<R1> && is_nothrow_move_assignable_v<D>`.

Requires: `(is_nothrow_move_assignable_v<R1> || is_copy_assignable_v<R1>)` is `true` and `(is_nothrow_move_assignable_v<D> || is_copy_assignable_v<D>)` is `true`.

Effects: Equivalent to

```
reset();
if constexpr (is_nothrow_move_assignable_v<R1>) {
    if constexpr (is_nothrow_move_assignable_v<D>) {
        resource = std::move(rhs.resource);
        deleter  = std::move(rhs.deleter);
    } else {
        deleter  = rhs.deleter;
        resource = std::move(rhs.resource);
    }
} else {
    if constexpr (is_nothrow_move_assignable_v<D>) {
        resource = rhs.resource;
        deleter  = std::move(rhs.deleter);
    } else {
        resource = rhs.resource;
        deleter  = rhs.deleter;
    }
}
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*]

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

7.5.7 `unique_resource` destructor [scope.unique_resource.dtor]

```
~unique_resource();
```

Effects: Equivalent to `reset()`.

7.5.8 `unique_resource` member functions [scope.unique_resource.mfun]

`void reset() noexcept;`

1 *Effects:* Equivalent to:

```
if (execute_on_destruction) {
    execute_on_destruction = false;
    if constexpr ( is_reference_v<R> )
        deleter(resource.get());
    else
        deleter(resource);
}
```

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

2 *Requires:* The expression `deleter(r)` shall be well-formed, shall have well-defined behavior, and shall not throw an exception.

3 *Remarks:* This function `reset` shall not participate in overload resolution if the selected assignment expression statement assigning `resource` is ill-formed.

4 *Effects:* Equivalent to:

```
reset();
if constexpr ( is_nothrow_assignable_v<R1&,RR> )
    resource = std::forward<RR>(r);
else
    resource = as_const(r);
execute_on_destruction = true;
```

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

`void release() noexcept;`

5 *Effects:* Equivalent to `execute_on_destruction = false`.

`const R& get() const noexcept;`

6 *Returns:* `resource`.

see below `operator*() const noexcept;`

7 *Requires:* The return type is equivalent to `add_lvalue_reference_t<remove_pointer_t<R>>`.

8 *Remarks:* This operator shall not participate in overload resolution unless `is_pointer_v<R> && !is_void_v<remove_pointer_t<R>>` is `true`.

9 *Effects:* Equivalent to:

```
return *get();
```

`R operator->() const noexcept;`

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

11 *Returns:* `get()`.

`const D & get_deleter() const noexcept;`

12 *Returns:* `deleter`.

7.5.9 Factory for `unique_resource` [scope.make_unique_resource]

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

1 *Requires:* The expression `( resource == invalid ? true : false )` shall be well-formed, have well-defined behavior, and shall not throw an exception.

2 *Effects:* Constructs and returns `unique_resource{std::forward<R>(resource), std::forward<D>(d)}`.

3 If `bool(resource == invalid)` evaluates to `true` before the construction, the returned `unique_resource` object's `execute_on_destruction` is `false`. In that case, any failure during the construction of the return value will not call `d(resource)`.

4 [*Note:* This factory function exists to avoid calling a deleter function with an invalid argument. — *end note*]

5 [*Example:* The following example shows its use to avoid calling `fclose` when `fopen` fails

```
auto file = make_unique_resource_checked(
  ::fopen("potentially_nonexistent_file.txt", "r"),
  nullptr, &::fclose);
```

— *end example*]

7.5.10 Feature test macro

For the purposes of SG10, we recommend the feature-testing macro name `__cpp_lib_scope`.

8 Appendix: Example Implementation

See https://github.com/PeterSommerlad/SC22WG21_Papers/tree/master/workspace/P0052_scope_exit/src