D3949 - Scoped Resource - Generic RAII Wrapper for the Standard Library

Peter Sommerlad and Andrew L. Sandoval

2014-02-28

Document Number: D3949	(update of N3830, N3677)
Date:	2014-02-28
Project:	Programming Language C++

1 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.

2 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

3 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.

4 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.
- Thanks also to members of the mailing lists who gave feedback. Especially Zhihao Yuan, and Ville Voutilainen.
- Special thanks to Daniel Krgler for his deliberate review of the draft version of this paper (D3949).

5 Motivation and Scope

The quality of C++ code can often be improved through the use of "smart" container objects. For example, using `std::unique_ptr` or `std::shared_ptr` to manage pointers can prevent common mistakes that lead to memory leaks, as well as the less common leaks that occur when exceptions unwind. The latter case is especially difficult to diagnose and debug and is a commonly made mistake – especially on systems where unexpected events (such as access violations) in third party libraries may cause deep unwinding

that a developer did not expect. (One example would be on Microsoft Windows with Structured Exception Handling and libraries like MFC that issue callbacks to user-defined code wrapped in a `try/catch(...)` block. The developer is usually unaware that their code is wrapped with an exception handler that depending on compile-time options will quietly unwind their code, masking any exceptions that occur.)

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

This proposal introduces a new RAII "smart" resource container called `unique_resource_t` which can bind a resource to "clean-up" code regardless of type of the argument required by the "clean-up" function.

5.1 Without Coercion

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

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

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

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

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

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

```
                        FILE_ALL_ACCESS,
                        FILE_SHARE_READ,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        nullptr),
                CloseHandle);

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

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

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

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

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

        // Read some data using the handle
        std::array<char, 1024> arr = { };
```

```
        DWORD dwRead = 0;
        // cast operator makes it seamless to use with other APIs needing a HANDLE
        ReadFile(hFile,
                &arr[0],
                static_cast<DWORD>(arr.size()),
                &dwRead);
}
```

5.1.1 Non-Pointer Handle Types

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

```
void demontrate_unique_resource_with_POSIX_IO(){
        const char* const filename = "hello.txt";
        {
                auto file = unique_resource_checked(::open(filename,O_CREAT|O_RDWR),
                                -1,&::close);
                ::write(file,"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());
        {
                auto file = unique_resource_checked(::open("nonexistingfile.txt", O_RDONLY),
                                -1, &::close);
                ASSERT_EQUAL(-1,file.get());
        }

}

void demontrate_unique_resource_with_POSIX_IO(){
        const char* const filename = "hello1.txt";
        {
                auto file = unique_resource_checked(
                                ::open(filename,O_CREAT|O_RDWR),
                                -1,&::close);
                ::write(file,"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);
        {
                auto file = unique_resource_checked(
                                ::open("nonexistingfile.txt", O_RDONLY),
                                -1,
                                &::close);
                ASSERT_EQUAL(-1,file.get());
        }
}
```

5.2 Multiple Parameters

This feature was abandoned due to feedback by LEWG in Issaquah. A lambda as deleter can have the same effect without complicating `unique_resource_t`.

5.3 Lambdas, etc.

It is also possible to use lambdas instead of a function pointer to initialize a `unique_resource`. The following is a very simple and otherwise useless example:

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


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

```
}


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

The example also shows that a scope guard can be released early (that is the clean-up function is not called).

5.4 Other Functionality

In addition to the basic features shown above, `unique_resource_t` also provides various operators (cast, `->`, `()`, `*`, and accessor methods (`get`, `get_deleter`). The most complicated of these is the `invoke()` member function which allows the "clean-up" function to be executed early, just as it would be at scope exit. This function takes a parameter indicating whether or not the function should again be executed at scope exit. The `reset(R&& resource)` member function that allows the resource value to be reset.

As already shown in the examples, the expected method of construction is to use one of the two generator functions:

- `unique_resource(resources,deleter)` - non-checking instance, allows multiple parameters.

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

5.5 What's not included

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

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

6 Impact on the Standard

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

7 Design Decisions

7.1 General Principles

The following general principles are formulated for `unique_resource_t`, and are valid for `scope_guard_t` correspondingly.

- Simplicity - Using `unique_resource_t` 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_t` object does. By binding the resource to it's clean-up routine, the declaration of `unique_resource_t` makes its intention clear.
- Resource Conservation and Lifetime Management - Using `unique_resource_t` makes it possible to "allocate it and forget about it" in the sense that deallocation is always accounted for after the `unique_resource_t` has been initialized.
- Exception Safety - Exception unwinding is one of the primary reasons that `unique_resource_t` is needed. Nevertheless the goal is to introduce a new container that will not throw during construction of the `unique_resource_t` itself. However, there are no intentions to provide safeguards for piecemeal construction of resource and deleter. If either fails, no unique_resource_t 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.

7.2 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` 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

7.3 Open Issues to be Discussed

- Should there be a companion class for sharing the resource `shared_resource` ? (Peter thinks no. Ville thinks it could be provided later anyway.)
- Should the proposed scope guard mechanism and unique resource go into (a) different header(s)?
- Should `scope_guard_t()` and `unique_resource::invoke()` guard against deleter functions that throw with `try deleter(); catch(...)` (as now) or not?
- Does `scope_guard_t` need to be move-assignable? Peter doesn't think so.

8 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::`.

8.1 Header

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

Table 1: Table 44 - General utilities library summary

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

8.2 Additional sections

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

8.3 Scope Guard [utilities.scope_guard]

This subclause contains infrastructure for a generic scope guard.

Header <scope_guard> synopsis

1 The header <scope_guard> defines the class template scope_guard_t and the function template scope_guard() to create its instances.

```
namespace std {
template <typename D>
struct scope_guard_t {
    // construction
    explicit
        scope_guard_t(D &&f) noexcept;
        // clean up
        ~scope_guard_t();
        // early release
        void release() noexcept { execute_on_destruction=false;}
        // move
        scope_guard_t(scope_guard_t  &&rhs) noexcept;

private:
        scope_guard_t(scope_guard_t const &)=delete;
        void operator=(scope_guard_t const &)=delete;
        scope_guard_t& operator=(scope_guard_t &&)=delete;
        D deleter; // exposition only
        bool execute_on_destruction; // exposition only
};
// factory function
template <typename D>
scope_guard_t<D> scope_guard(D && deleter) noexcept {
        return scope_guard_t<D>{std::move(deleter)};
}

} // namespace std
```

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

8.3.1 Class Template scope_guard_t [scope_guard.scope_guard_t]

1 *Requires:* D shall be a MoveConstructible function object type or reference to such, the expression deleter() shall be valid and shall not throw an exception. Move construction of D shall not throw an exception.

```
explicit
scope_guard_t(D &&deleter) noexcept;
```

2 *Effects:* constructs a scope_guard_t object that will call deleter() on its destruction if not release()ed prior to that. execute_on_destruction is set to true.

```
~scope_guard_t();
```
3 *Effects:* If and only if `execute_on_destruction` is `true`, calls `deleter()`.

```
void release() noexcept;
```
4 *Effects:* `execute_on_destruction=false;`

```
scope_guard_t(scope_guard_t  &&rhs) noexcept;
```
5 *Effects:* Move constructs `deleter` from `rhs.deleter`. `execute_on_destruction=rhs.execute_on_destruction;rhs.release();`

8.3.2 Factory Function `scope_guard` [scope_guard.scope_guard_]

```
template <typename D>
scope_guard_t<D> scope_guard(D && deleter) noexcept;
```
1 *Returns:* `scope_guard_t<D>(std::move(deleter))`

8.4 Unique Resource Wrapper [utilities.unique_resource]

This subclause contains infrastructure for a generic RAII resource wrapper.

Header `<unique_resource>` synopsis

1 The header `<unique_resource>` defines the class template `unique_resource_t`, the enumeration `invoke_it` and function templates `unique_resource()` and `unique_resource_checked()` to create its instances.

```
namespace std {
enum class invoke_it { once, again };

template<typename R,typename D>
class unique_resource_t {
        R resource; // exposition only
        D deleter; // exposition only
        bool execute_on_destruction; // exposition only
        unique_resource_t& operator=(unique_resource_t const &)=delete;
        unique_resource_t(unique_resource_t const &)=delete;
public:
        // construction
        explicit
        unique_resource_t(R && resource, D && deleter, bool shouldRun=true) noexcept;
        // move
        unique_resource_t(unique_resource_t &&other) noexcept;
        unique_resource_t& operator=(unique_resource_t  &&other) noexcept ;

        // resource release
        ~unique_resource_t() ;
        void invoke(invoke_it const strategy = invoke_it::once) noexcept ;
        R&& release() noexcept;
        void reset(R && newresource) noexcept ;
        // resource access
```

```
        R const & get() const noexcept ;
        operator  R const &() const noexcept ;
        R operator->() const noexcept ;
        see below operator*() const;
        // deleter access
        const D &       get_deleter() const noexcept;
        D &     get_deleter() noexcept;
};

//factories
template<typename R,typename D>
unique_resource_t<R,D>
unique_resource( R && r,D t) noexcept;
template<typename R,typename D>
unique_resource_t<R,D>
unique_resource_checked(R r, R invalid, D t ) noexcept;

} // namespace std
```

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

8.4.1 Class Template `unique_resource_t` [unique_resource.unique_resource_t]

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

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

2 *Effects:* constructs a `unique_resource_t` by moving `resource` and then `deleter`. The constructed object will call `deleter(resource)` on its destruction if not `release()`ed prior to that. `execute_on_destruction` is set to `true`.

```
unique_resource_t(unique_resource_t &&other) noexcept;
```

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

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

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

```
~unique_resource_t() ;
```

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

```
void invoke(invoke_it const strategy = invoke_it::once) noexcept;
```

6 *Effects:*

```
  if (execute_on_destruction) try {
          this->get_deleter()(resource);
  } catch(...){}
  execute_on_destruction=(strategy==invoke_it::again);
```

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

7 *Effects:* `execute_on_destruction=false;`

8 *Returns:* `resource`

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

9 *Effects:*

```
  this->invoke(invoke_it::again);
  resource=std::move(newresource);
```

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

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

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

12 *Returns:* `resource`.

see below `operator*() const noexcept;`

13 *Requires:* This function is only available if `is_pointer<R>::value` is `true`.

14 *Returns:* `*this->get()`.
Return type is `std::add_lvalue_reference_t<std::remove_pointer_t<R>>`

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

15 *Returns:* `deleter`

8.4.2 Factories for `unique_resource_t` [unique_resource.unique_resource]

```
template<typename R,typename D>
unique_resource_t<R,D>
unique_resource( R && r,D t) noexcept ;
```

1 *Returns:* `unique_resource_t<R,D>(std::move(r), std::move(t),true);`

```
template<typename R,typename D>
unique_resource_t<R,D>
unique_resource_checked(R r, R invalid, D t ) noexcept;
```

2 *Requires:* `R` is EqualityComparable

3 *Returns:* `unique_resource_t<R,D>(std::move(r), std::move(t), not bool(r==invalid);`

9 Appendix: Example Implementations

9.1 Scope Guard

```
#ifndef SCOPE_GUARD_H_
#define SCOPE_GUARD_H_

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

namespace std{
namespace experimental{

template <typename D>
struct scope_guard_t {
        explicit
        scope_guard_t(D &&f) noexcept
        :deleter(std::move(f))
        ,execute_on_destruction{true}{}
        ~scope_guard_t(){
                if (execute_on_destruction)
                        try{
                                deleter();
                        }catch(...){}
        }
        void release() noexcept { execute_on_destruction=false;}
        scope_guard_t(scope_guard_t  &&rhs) noexcept
        :deleter(std::move(rhs.deleter))
        ,execute_on_destruction{rhs.execute_on_destruction}{
                rhs.release();
        }

private:
        scope_guard_t(scope_guard_t const &)=delete;
        void operator=(scope_guard_t const &)=delete;
        scope_guard_t& operator=(scope_guard_t &&)=delete;
        D deleter;
        bool execute_on_destruction; // exposition only
};

// usage: auto guard=scope_guard([] std::cout ¡¡ "done.";);
template <typename D>
scope_guard_t<D> scope_guard(D && deleter){
        return scope_guard_t<D>(std::move(deleter)); // fails with curlies
}
```

```
}
}

#endif /* SCOPE_GUARD_H_ */
```

9.2 Unique Resource

```
#ifndef UNIQUE_RESOURCE_H_
#define UNIQUE_RESOURCE_H_

namespace std{
namespace experimental{
enum class invoke_it { once, again };

template<typename R,typename DELETER>
class unique_resource_t {
        R resource;
        DELETER deleter; // deleter must be void(R) noexcept compatible
        bool execute_on_destruction; // exposition only
        unique_resource_t& operator=(unique_resource_t const &)=delete;
        unique_resource_t(unique_resource_t const &)=delete; // no copies!
public:
        // construction
        explicit
        unique_resource_t(R && resource, DELETER && deleter, bool execute_on_destruction=true) noe
                :  resource(std::move(resource))
                ,  deleter(std::move(deleter))
                , execute_on_destruction{execute_on_destruction}{}
        // move
        unique_resource_t(unique_resource_t &&other) noexcept
        :resource(std::move(other.resource))
        ,deleter(std::move(other.deleter))
        ,execute_on_destruction{other.execute_on_destruction}{
                other.release();
        }
        unique_resource_t& operator=(unique_resource_t  &&other) noexcept {
                this->invoke(invoke_it::once);
                deleter=std::move(other.deleter);
                resource=std::move(other.resource);
                execute_on_destruction=other.execute_on_destruction;
                other.release();
                return *this;
        }
    // resource release
        ~unique_resource_t() {
                this->invoke(invoke_it::once);
        }
        void invoke(invoke_it const strategy = invoke_it::once) noexcept {
```

```
                if (execute_on_destruction) {
                        try {
                                this->get_deleter()(resource);
                        } catch(...){}
                }
                execute_on_destruction = strategy==invoke_it::again;
        }
        R const & release() noexcept{
                execute_on_destruction = false;
                return this->get();
        }
        void reset(R && newresource) noexcept {
                invoke(invoke_it::again);
                resource = std::move(newresource);
        }

        // resource access
        R const & get() const noexcept {
                return resource;
        }
        operator  R const &() const noexcept {
                return resource;
        }
        R operator->() const noexcept {
                return resource;
        }
        std::add_lvalue_reference_t<
                std::remove_pointer_t<R>>
        operator*() const {
                return * resource;
        }

        // deleter access
        const DELETER &
        get_deleter() const noexcept {
                return deleter;
        }
};

//factories
template<typename RES,typename DELETER>
unique_resource_t<RES,DELETER>
unique_resource( RES && r,DELETER t) {
        return unique_resource_t<RES,DELETER>(std::move(r), std::move(t),true);
}

template<typename RES,typename DELETER>
unique_resource_t<RES,DELETER>
unique_resource_checked(RES r, RES invalid, DELETER t ) {
```

```
        bool execute_on_destruction=(r != invalid);
        return unique_resource_t<RES,DELETER>(std::move(r), std::move(t), execute_on_destruction);
}

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