D3830 - Scoped Resource - Generic RAII Wrapper for the Standard Library

Peter Sommerlad and Andrew L. Sandoval

2013-12-23

Document Number:	D3830 (update of N3677)
Date:	2013-12-23
Project:	Programming Language C++

1 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.

2 Introduction

The Standard Template Library provides RAII classes for managing pointer types, such as `std::unique_ptr` and `std::shared_ptr`. This proposal seeks to add a new generic RAII class which ties zero or more resources to a clean-up/completion routine which is bound by scope, ensuring execution at scope exit (as the object is destroyed) unless released or executed early.

3 Acknowledgements

- 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 fullfilling 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 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.

- This proposal refers to N3829 for the implementation of `std::apply()`. It should be voted in after N3829, because its description relies on the existence of `apply()` in the standard library. However, it can also be applied independently with appropriate changes to the specification.

4 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.)

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

4.1 Without Coercion

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

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

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

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

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

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

        // Read some data using the handle
        std::array<char, 1024> arr = { };
        DWORD dwRead = 0;
        ReadFile(hFile.get(),    // Must use hFile.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 `scoped_resource` as shown in the following example:

```
void ScopedResourceExample1()
{
        // Initialize hFile ensure it will be "closed" (regardless of value) on scope-exit
        auto hFile = std::make_scoped_resource(
                CloseHandle,                            // Clean-up API
                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!

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

4.2 Multiple Parameters

In some cases it is convenient to invoke a "clean-up" API that requires multiple parameters. While this could be done using `std::bind`, it is not necessary with `std::scoped_resource`. Variadic template parameters and `std::tuple` make it possible to bind more than one value to a resource's clean-up function. In the example shown below, the system defined clean-up function `VirtualFree` requires 3 parameters, only one of which is the "resource" (which in this case is a block of virtual memory.)

```
// System defined APIs:
void* VirtualAlloc(
    void *pAddress,
    size_t size,
    unsigned int fAllocationType,
    unsigned int fProtect);

bool VirtualFree(
    void *pAddress,
    size_t size,
    unsigned int uiFreeType);

void ScopedResourceMultiExample()
{
        auto pVirtualMem = make_scoped_resource(
                VirtualFree,                    // Clean-up function
                VirtualAlloc(nullptr,
                        PAGE_SIZE,
                        MEM_COMMIT,
                        PAGE_READWRITE),        // Resource (1st parameter)
                0,                              // 2nd parameter to VirtualFree
                MEM_RELEASE);                   // 3rd Parameter to VirtualFree

        if(nullptr == pVirtualMem)
        {
                // Handle error...
        }
        .
        .
        .
}
```

In this example, `VirtualFree` will be invoked with all three required parameters on scope exit. Though useless in this example, it is also possible to access the other parameters. The cast operator will always cast to the type of and return the first parameter (the real resource in this example), but the `get<T>()` method allows access to the other parameters as in this example:

```
if(pVirtualMem.get<2>() != MEM_RELEASE)
{
        // Something is not right!
}
```

4.3 Lambdas, etc.

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

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

        auto altgoodbye = std::make_scoped_resource([](const std::string &strBye) ->void
        {
                std::cout << strFarewell.c_str() << std::endl;
        },
        strFarewell);
        if(false == strFarewell.empty())
        {
                goodbye.release();              // Don't use goodbye!
        }
        else
        {
                altgoodbye.release();    // Don't use the alternate
        }
        // Say hello first!
        std::cout << "Hello World" << std::endl;
}
```

Though there is obviously no need for a `scoped_resource` for something this simple, the example not only shows that a lambda may be used in place of a function, but also that no "resource value" is required if the "clean-up" function takes no parameters. The example also shows that a resource can be released (that is the clean-up function is not called). Methods also exist to allow it to execute earlier than at scope exit.

4.4 Other Functionality

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

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

- `make_scoped_resource(deleter, resources...)` - non-checking instance, allows multiple parameters.
- `make_scoped_resource_checked(deleter, resource, invalid-value)` - checked instance, allowing only one resource which is validated prior to invoking the deleter function.

4.5 What's not included

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

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

5 Impact on the Standard

This proposal is a pure library extension. A new header, `<scope_resource>` is proposed, but it does not require changes to any standard classes or functions. It does not require any changes in the core language, and it has been implemented in standard C++ conforming to c++14.

6 Design Decisions

6.1 General Principles

- Simplicity - Using `scoped_resource` should be nearly as simple as using an unwrapped type. The generator functions, cast operator, and accessors all enable this.
- Transparency - It should be obvious from a glance what each instance of a `scoped_resource` object does. By binding the resource to it's clean-up routine, the declaration of `scoped_resource` makes its itention clear.
- Resource Conservation and Lifetime Management - Using `scoped_resource` makes it possible to "allocate it and forget about it" in the sense that deallocation is always accounted for after the `scoped_resource` has been initialized.
- Exception Safety - Exception unwinding is one of the primariy reasons that `scoped_resource` is needed. Nevertheless the goal is to introduce a new container that will not throw during construction of the `scoped_resource` itself.
- Flexibility - `scoped_resource` is designed to be flexible, allowing the use of lambdas or existing functions for clean-up of resources, as well as allowing for zero or more parameters. This is a vast improvement over the originally proposed classes in N3677.

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 `scoped_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.

6.3 Open Issues to be Discussed

- Should `scoped_resource` be move-able like `unique_ptr` ?
- Should there be a companion class for sharing the resource `shared_resource` ?
- Is the set of provided overloaded operators really useful ? Which could be removed without harm ? Is there something missing ?
- Is scoped_resource lightweight enough, or should the mechanics for early-release and re-assignment be omitted ?

7 Technical Specifications

7.1 Header

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

```
Table 44 - General utilities library summary
---------------------------------------------
|Subclause             |  Header(s)         |
---------------------------------------------
|20.14 Scoped Resource | <scoped_resource>  |
---------------------------------------------
```

7.2 Additional section

Add a new section to chapter 20 introducing the contents of the header `scoped_-resource`.

7.3 Scoped Resource Wrapper [utilities.scoped_resource]

This subclause contains infrastructure for a generic RAII resource wrapper.

Header `<scoped_resource>` synopsis

1 The header `<scoped_resource>` defines the class template `scoped_resource`, the enumeration invoke_it and function templates to create its instances.

```
namespace std {
enum class invoke_it
{
        once,
        again
};
template<typename DELETER, typename ... R>
class scoped_resource
{
        DELETER deleter; // exposition only
        std::tuple<R...> resource; // exposition only
        bool execute_on_destruction; // exposition only
        scoped_resource& operator=(scoped_resource const &)=delete;
        scoped_resource(scoped_resource const &)=delete;
public:
        // construction
        explicit
        scoped_resource(DELETER&& deleter, R&&... resource, bool shouldRun = true) noexcept;
        explicit
        scoped_resource(const DELETER& deleter, const R&... resource, bool shouldRun=true) noexcep
        // move
        scoped_resource(scoped_resource &&other) noexcept;
        scoped_resource& operator=(scoped_resource  &&other) noexcept;

        // resource release
        ~scoped_resource();
        void invoke(invoke_it const strategy = invoke_it::once) noexcept;
        see below release() noexcept;
        void reset(R... newresource) noexcept;

        // resource accessors
        operator  see below() const noexcept; // resource type cast

        template<size_t n = 0>
        see below  get() const noexcept;
        see below  operator*() const noexcept;
        see below  operator->() const noexcept;

        // deleter access
        const DELETER & get_deleter() const noexcept;

};
// factory functions
template<typename DELETER, typename ... R>
        auto make_scoped_resource(DELETER t, R ... r) {
        return scoped_resource<DELETER, R...>(std::move(t), std::move(r)...);
}

template<typename DELETER, typename RES>
auto make_scoped_resource_checked(DELETER t, RES r, RES invalid) {
        auto shouldrun=(r != invalid);
        return scoped_resource<DELETER, RES>(std::move(t), std::move(r), shouldrun);
}
} // namespace std
```

2 [*Note:* `scoped_resource` is meant to be a universal RAII wrapper for resource handles provided by an operating system or platform. Typically, such resource handles come with

a factory function and a release function and are of trivial type. The release function together with the result of the factory function is used to create a scoped_resource 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*]

7.3.1 Class Template `scoped_resource` [scoped_resource.scoped_resource]

```
explicit
scoped_resource(DELETER&& deleter, R&&... resource,
                bool shouldRun = true) noexcept;
explicit
scoped_resource(const DELETER& deleter, const R&... resource,
                bool shouldRun = true) noexcept;
```

1 *Requires:* `deleter` is a `noexcept` function (object) taking `resource...` as arguments.

2 *Effects:* construct a scoped_resource that will call `deleter(resource...)` on its destruction if `shouldRun` is true.

```
        scoped_resource(scoped_resource &&other) noexcept;
```

3 *Effects:* move construct a scoped_resource by moving members. Sets `other.execute_on_destruction` to false.

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

4 *Effects:* `this->invoke();` moves all members from other. Sets `other.execute_on_destruction` to false.

```
        ~scoped_resource();
```

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

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

6 *Effects:*

```
  if (execute_on_destruction)
      apply(deleter,resource);
  execute_on_destruction=(strategy==invoke_it::again);
```

```
        see below release() noexcept;
```

7 *Effects:* `execute_on_destruction=false;`

8 *Returns:* `resource`.
Return type is `tuple<R...>`, if `sizeof(R...)>1` or `tuple_element_t<0,tuple<R...>>` if `sizeof(R...)==1`, or `void`.

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

9 *Effects:*

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

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

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

11 *Returns:* `get<0>(resource)`.
Return type and `operator` name is the underlying type `tuple_element_t<0,tuple<R...>>`.

```
template<size_t n = 0>
see below  get() const noexcept;
```

12 *Returns:* `get<n>(resource)`.
Return type is `tuple_element_t<n,tuple<R...>> const &`.
[*Note:* An implementation might chose `auto const &` return type deduction. — *end note*]

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

13 *Requires:* These functions are only available if `is_pointer<tuple_element_t<0,tuple<R...>>>::value` is `true`. The second one only applies if the return type is a class or union type.

14 *Returns:* `*get<0>(resource)`
Return type is `decltype(*get<0>(resource))`.

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

15 *Returns:* `deleter`

8 Example Implementation

```
// Copyright Peter Sommerlad and Andrew L. Sandoval 2012 - 2013.
// Distributed under the Boost Software License, Version 1.0.
// (See accompanying file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

//
// scoped_resource - sample implementation
// Written by Professor Peter Sommerlad
// In response to N3677 (http://www.andrewlsandoval.com/scope_exit/)
// which was discussed in the Fall 2013 ISO C++ LEWG committee meeting.
//
// scoped_resource is a more versatile replacement for the 4 RAII classes
// presented in N3677.
//
// Subsequent (editing, etc.) work by:
// -Andrew L. Sandoval


#ifndef SCOPED_RESOURCE_H_
#define SCOPED_RESOURCE_H_
#include <cstddef>
#include <tuple>
```

```
#include <type_traits>
#include <utility>

// should actually be part of the standard, if it isn't already.
namespace apply_ns{
using namespace std;
// from C++14 standard draft example, doesn't work perfect, but fixable here
// fixed by removing reference from std::tuple_size¡Tuple¿
template<typename F, typename Tuple, size_t ... I>
auto apply_impl(F&& f, Tuple&& t, index_sequence<I...>)
//noexcept(noexcept(forward¡F¿(f)(get¡I¿(forward¡Tuple¿(t))...)))
{
        return forward<F>(f)(get<I>(forward<Tuple>(t))...);
}
template<typename F, typename Tuple>
auto apply(F&& f, Tuple&& t)
//noexcept(noexcept(apply_impl(forward¡F¿(f), forward¡Tuple¿(t), make_index_sequence¡tuple_-
size¡decay_t¡Tuple¿¿::value¿)))
{
  using Indices = make_index_sequence<tuple_size<decay_t<Tuple>>::value>;
  return apply_impl(forward<F>(f), forward<Tuple>(t), Indices{});
}
}

// something like the following should be available in the standard, if not it should
// tuple_element gets problems with the empty case...
namespace select_first{
// the following idea doesn't work, because it instantiates tuple_element with an empty tuple
/*
template <typename ... L>
using  first_type =std:: conditional< sizeof ...( L)!=0,typename std:: tuple_element<0,std:: tuple<L...>>::ty
 */

template <typename ... L> struct first_type_or_void;
template <typename F, typename ...L>
struct first_type_or_void<F,L...>{
        using type=F;
};
template <> struct first_type_or_void<>{
        using type=void;
};
template <typename ...L>
using first_type_or_void_t = typename first_type_or_void<L...>::type;
}

// shouldn't be e member type of scoped_resource, because it will be impossible to spell
enum class invoke_it { once, again };
// provide it as variadic template
template<typename DELETER, typename ... R>
```

```
class scoped_resource {
        DELETER deleter; // deleter must be void(R...) noexcept compatible
        std::tuple<R...> resource;
        bool execute_on_destruction;
        scoped_resource& operator=(scoped_resource const &)=delete;
        scoped_resource(scoped_resource const &)=delete; // no copies!
        auto do_release(std::true_type is_size_one){
                execute_on_destruction = false;
                return std::get<0>(std::tuple_cat(resource,std::tuple<bool>{}));
                // avoid compile errors if sizeof(R...) is 0.
        }
        auto do_release(std::false_type is_some_other_size){
                execute_on_destruction = false;
                return resource;
        }
public:
        scoped_resource(scoped_resource &&other)
        :deleter{std::move(other.deleter)}
        ,resource{std::move(other.resource)}
        ,execute_on_destruction{other.execute_on_destruction}{
                other.execute_on_destruction=false;
        }
        scoped_resource& operator=(scoped_resource  &&other){
                this->invoke();
                deleter=std::move(other.deleter);
                resource=std::move(other.resource);
                execute_on_destruction=other.execute_on_destruction;
                other.execute_on_destruction = false;
                return *this;
        }
        explicit
        scoped_resource(DELETER deleter, R... resource, bool shouldrun=true) noexcept
                : deleter{std::move(deleter)}
                , resource{std::make_tuple(std::move(resource)...)}
                , execute_on_destruction{shouldrun}{}
        ~scoped_resource() {
                invoke(invoke_it::once);
        }

        auto release(){
                return do_release(std::conditional_t<sizeof...(R)==1,
                             std::true_type,std::false_type>{});
        }

        // reset – only resets the resource, not the deleter!
        void reset(R... newresource) noexcept {
                invoke(invoke_it::again);
                resource = std::make_tuple(std::move(newresource)...);
        }
        // get - for cases where cast operator is undesirable (e.g. as a ... parameter)
        template<size_t n = 0>
        auto const & get() const noexcept {
                return std::get<n>(resource);
        }
        //
        // cast operator
        // provide that for the single/first value case
        operator  select_first::first_type_or_void_t<R...>() const noexcept {
                return get();
        }

        // ?? are the following operators useful?
        // operator-¿ for accessing members on pointer types

        std::add_lvalue_reference_t<
                std::remove_pointer_t<
                        select_first::first_type_or_void_t<R...>>>
        operator->() const {
                return std::get<0>(resource);
        }

        //
        // operator* for dereferencing pointer types

        std::add_lvalue_reference_t<
                std::remove_pointer_t<
                        select_first::first_type_or_void_t<R...>>>
        // auto const & should be better... but clang crashes
        operator*() const {
                return *get();          // If applicable
        }
        const DELETER& get_deleter() const noexcept {
                return deleter;
        }
        void invoke(invoke_it const strategy = invoke_it::once) noexcept {
                if (execute_on_destruction) {
                        apply_ns::apply(deleter, resource);
                }
                execute_on_destruction = strategy==invoke_it::again;
        }
};

template<typename DELETER, typename ... R>
        auto make_scoped_resource(DELETER t, R ... r) {
        return scoped_resource<DELETER, R...>(std::move(t), std::move(r)...);
}
// the following convenience factory is intended for a single resource only, i.e. file descriptor
template<typename DELETER, typename RES>
```

```
auto make_scoped_resource_checked(DELETER t, RES r, RES invalid) {
        auto shouldrun=(r != invalid);
        return scoped_resource<DELETER, RES>(std::move(t), std::move(r), shouldrun);
}


#endif /* SCOPED_RESOURCE_H_ */
```