Boost.ScopeExit 1.1.0

Alexander Nasonov

Lorenzo Caminiti <lorcaminiti@gmail.com>

Table of Contents

This library allows to execute arbitrary code when the enclosing scope exits.

Introduction

Nowadays, every C++ developer is familiar with the Resource Acquisition Is Initialization (RAII) technique. It binds resource acquisition and release to initialization and destruction of a variable that holds the resource. There are times when writing a special class for such a variable is not worth the effort. This is when Boost.ScopeExit comes into play.

Programmers can put resource acquisition directly in their code and next to it, they can write code that releases the resource using this library. For example (see also `world.cpp`): [1]

```
void world::add_person(person const& a_person) {
    bool commit = false;

    persons_.push_back(a_person);           // (1) direct action
    // Following block is executed when the enclosing scope exits.
    BOOST_SCOPE_EXIT(&commit, &persons_) {
        if(!commit) persons_.pop_back();    // (2) rollback action
    } BOOST_SCOPE_EXIT_END

    // ...                                  // (3) other operations

    commit = true;                          // (4) disable rollback actions
}
```

[1] Older versions of this library used a Boost.Preprocessor sequence to specify the list of captured variables. While maintaining full backward compatibility, it is now possible to specify the captured variables also using a comma-separated list (which is the preferred syntax). See the No Variadic Macros section for more information.

Getting Started

This section explains how to setup a system to use this library.

This Documentation

Programmers should have enough knowledge to use this library after reading the Introduction, Getting Started, and Tutorial sections. The Reference section can be consulted at a later point for quick reference. All the other sections of this documentation can be considered optional.

Some footnotes are marked by the word "**Rationale**". They explain reasons behind decisions made during the design and implementation of this library.

In most of the examples presented in this documentation, the Boost.Detail/LightweightTest (`boost/detail/lightweight_test.hpp`) macro `BOOST_TEST` is used to check correctness conditions. The `BOOST_TEST` macro is conceptually similar to `assert` but a failure of the checked condition does not abort the program, instead it makes `boost::report_errors` return a non-zero program exit code. [2]

Compilers and Platforms

The authors originally developed and tested the library on GNU Compiler Collection (GCC) C++ 3.3, 3.4, 4.1, 4.2, 4.5.3 (with and without C++11 features `-std=c++0x`), Microsoft Visual C++ (MSVC) 8.0, and Intel 10.1 under Linux, Cygwin, and Windows 7. However, this library should be usable on any compiler that supports Boost.Typeof except:

- MSVC 7.1 and 8.0 fail to link if a function with Boost.ScopeExit is included by multiple translation units.
- GCC 3.3 cannot compile Boost.ScopeExit inside a template (see http://lists.boost.org/Archives/boost/2007/02/116235.php for details).

See the library regression test results for detailed information on supported compilers and platforms. Check the library regression test `Jamfile.v2` for any special configuration that might be required for a specific compiler.

Installation

This library is composed of header files only. Therefore there is no pre-compiled object file which needs to be installed. Programmers can simply instruct the compiler where to find the library header files (`-I` option on GCC, `/I` option on MSVC, etc) and compile code using the library.

The library implementation uses Boost.Typeof to automatically deduce the types of the Boost.ScopeExit captured variables (see the Tutorial section). In order to compile code in type-of emulation mode, all types should be properly registered using `BOOST_TYPEOF_REGISTER_TYPE` and `BOOST_TYPEOF_REGISTER_TEMPLATE`, or appropriate Boost.Typeof headers should be included (see the source code of most examples presented in this documentation).

[2] **Rationale.** Using Boost.Detail/LightweightTest allows to add the examples to the library regression tests so to make sure that they always compile and run correctly.

Tutorial

This section illustrates how to use this library.

Capturing Variables

Imagine that we want to make many modifications to data members of some `world` class in its `world::add_person` member function. We start with adding a new `person` object to a vector of persons:

```
void world::add_person(person const& a_person) {
    bool commit = false;

    persons_.push_back(a_person);           // (1) direct action
    ...
```

Some operations down the road may throw an exception and all changes to involved objects should be rolled back. This all-or-nothing semantic is also known as strong guarantee.

In particular, the last added person must be deleted from `persons_` if the function throws. All we need is to define a delayed action (release of a resource) right after the direct action (resource acquisition). For example (see also `world.cpp`):

```
void world::add_person(person const& a_person) {
    bool commit = false;

    persons_.push_back(a_person);           // (1) direct action
    // Following block is executed when the enclosing scope exits.
    BOOST_SCOPE_EXIT(&commit, &persons_) {
        if(!commit) persons_.pop_back();    // (2) rollback action
    } BOOST_SCOPE_EXIT_END

    // ...                                  // (3) other operations

    commit = true;                          // (4) disable rollback actions
}
```

The block below point `(1)` is a Boost.ScopeExit declaration. Unlike point `(1)`, an execution of the Boost.ScopeExit body will be delayed until the end of the current scope. In this case it will be executed either after point `(4)` or on any exception. (On various versions of the GCC compiler, it is necessary to use `BOOST_SCOPE_EXIT_TPL` instead of `BOOST_SCOPE_EXIT` within templates, see later in this section for details.)

The Boost.ScopeExit declaration starts with the `BOOST_SCOPE_EXIT` macro invocation which accepts a comma-separated list of captured variables (a Boost.Preprocessor sequence is also accepted for compilers that do not support variadic macros and for backward compatibility with older versions of this library, see the No Variadic Macros section). If a capture starts with the ampersand sign `&`, a reference to the captured variable will be available inside the Boost.ScopeExit body; otherwise, a copy of the variable will be made after the Boost.ScopeExit declaration at point `(1)` and only the copy will be available inside the body (in this case, the captured variable's type must be `CopyConstructible`).

In the example above, the variables `commit` and `persons_` are captured by reference because the final value of the `commit` variable should be used to determine whether to execute rollback actions or not, and the action should modify the `persons_` object, not its copy. This is the most common case but passing a variable by value is sometimes useful as well.

Finally, the end of the Boost.ScopeExit body must be marked by the `BOOST_SCOPE_EXIT_END` macro which must follow the closing curly bracket `}` of the Boost.ScopeExit body.

> **Important**
>
> In order to comply with the STL exception safety requirements, the Boost.ScopeExit body must never throw (because the library implementation executes the body within a destructor call). This is true for all Boost.ScopeExit macros (including `BOOST_SCOPE_EXIT_TPL` and `BOOST_SCOPE_EXIT_ALL` seen below) on both C++03 and C++11.

Consider a more complex example where `world::add_person` can save intermediate states at some point and roll back to the last saved state. We use `person::evolution_` to store a version of the changes and increment it to cancel all rollback actions associated with those changes. If we pass a current value of `evolution_` stored in the `checkpoint` variable by value, it remains unchanged within the Boost.ScopeExit body so we can compare it with the final value of `evolution_`. If the latter was not incremented since we saved it, the rollback action inside the Boost.ScopeExit body should be executed. For example (see also `world_checkpoint.cpp`):

```
void world::add_person(person const& a_person) {
    persons_.push_back(a_person);

    // This block must be no-throw.
    person& p = persons_.back();
    person::evolution_t checkpoint = p.evolution;
    BOOST_SCOPE_EXIT(checkpoint, &p, &persons_) {
        if(checkpoint == p.evolution) persons_.pop_back();
    } BOOST_SCOPE_EXIT_END

    // ...

    checkpoint = ++p.evolution;

    // Assign new identifier to the person.
    person::id_t const prev_id = p.id;
    p.id = next_id_++;
    BOOST_SCOPE_EXIT(checkpoint, &p, &next_id_, prev_id) {
        if(checkpoint == p.evolution) {
            next_id_ = p.id;
            p.id = prev_id;
        }
    } BOOST_SCOPE_EXIT_END

    // ...

    checkpoint = ++p.evolution;
}
```

When multiple Boost.ScopeExit blocks are declared within the same enclosing scope, the Boost.ScopeExit bodies are executed in the reversed order of their declarations.

Capturing The Object `this`

Within a member function, it is also possible to capture the object `this`. However, the special symbol `this_` must be used instead of `this` in the Boost.ScopeExit declaration and body to capture and access the object. For example (see also `world_this.cpp`):

```
BOOST_SCOPE_EXIT(&commit, this_) { // Capture object `this_`.
    if(!commit) this_->persons_.pop_back();
} BOOST_SCOPE_EXIT_END
```

It is not possible to capture the object `this_` by reference because C++ does not allow to take a reference to `this`. If the enclosing member function is constant then the captured object will also be constant, otherwise the captured object will be mutable.

Capturing No Variable

A Boost.ScopeExit declaration can also capture no variable. In this case, the list of captured variables is replaced by the `void` keyword (similarly to the C++ syntax that allows to declare a function with no parameter using *`result-type function-name`*`(void)`). [3] For example, this can be useful when the Boost.ScopeExit body only needs to access global variables (see also `world_void.cpp`):

```
struct world_t {
    std::vector<person> persons;
    bool commit;
} world; // Global variable.

void add_person(person const& a_person) {
    world.commit = false;
    world.persons.push_back(a_person);

    BOOST_SCOPE_EXIT(void) { // No captures.
        if(!world.commit) world.persons.pop_back();
    } BOOST_SCOPE_EXIT_END

    // ...

    world.commit = true;
}
```

(Both compilers with and without variadic macros use this same syntax for capturing no variable, see the No Variadic Macros section for more information.)

Capturing All Variables (C++11 Only)

On C++11 compliers, it is also possible to capture all the variables in scope without naming them one-by-one using the special macro `BOOST_SCOPE_EXIT_ALL` instead of `BOOST_SCOPE_EXIT`. [4]

Following the same syntax adopted by C++11 lambda functions, the `BOOST_SCOPE_EXIT_ALL` macro accepts a comma-separated list of captures which must start with either `&` or `=` to capture all variables in scope respectively by reference or by value (note that no variable name is specified by these leading captures). Additional captures of specific variables can follow the leading `&` or `=` and they will override the default reference or value captures. For example (see also `world_checkpoint_all.cpp`):

[3] **Rationale.** Unfortunately, it is not possible to simply invoke the Boost.ScopeExit macro with no parameters as in `BOOST_SCOPE_EXIT()` because the C++ preprocessor cannot detect emptiness of a macro parameter when the parameter can start with a non-alphanumeric symbol (which is the case when capturing a variable by reference `&variable`).

[4] **Rationale.** The `BOOST_SCOPE_EXIT_ALL` macro is only defined on C++11 compilers for which the Boost.Config macro `BOOST_NO_CXX11_LAMBDAS` is not defined. Using `BOOST_SCOPE_EXIT_ALL` on C++03 compilers for which `BOOST_NO_CXX11_LAMBDAS` is defined will generate (possibly cryptic) compiler errors. Note that a new macro `BOOST_SCOPE_EXIT_ALL` needed to be introduced instead of reusing `BOOST_SCOPE_EXIT` because `BOOST_SCOPE_EXIT(&)` and `BOOST_SCOPE_EXIT(=)` cannot be distinguished from `BOOST_SCOPE_EXIT(void)` or `BOOST_SCOPE_EXIT(this_)` using the C++ preprocessor given that the symbols `&` and `=` are neither prefxied nor postfixed by alphanumeric tokens (this is not an issue for `BOOST_SCOPE_EXIT_ALL` which always has the non-alphanumeric `&` or `=` as the first capture so the first capture tokens are simply never compared with neither `void` nor `this_` for this macro).

```
void world::add_person(person const& a_person) {
    persons_.push_back(a_person);

    // This block must be no-throw.
    person& p = persons_.back();
    person::evolution_t checkpoint = p.evolution;
    // Capture all by reference `&`, but `checkpoint` and `this` (C++11 only).
    BOOST_SCOPE_EXIT_ALL(&, checkpoint, this) { // Use `this` (not `this_`).
        if(checkpoint == p.evolution) this->persons_.pop_back();
    }; // Use `;` (not `SCOPE_EXIT_END`).

    // ...

    checkpoint = ++p.evolution;

    // Assign new identifier to the person.
    person::id_t const prev_id = p.id;
    p.id = next_id_++;
    // Capture all by value `=`, but `p` (C++11 only).
    BOOST_SCOPE_EXIT_ALL(=, &p) {
        if(checkpoint == p.evolution) {
            this->next_id_ = p.id;
            p.id = prev_id;
        }
    };

    // ...

    checkpoint = ++p.evolution;
}
```

The first Boost.ScopeExit declaration captures all variables in scope by reference but the variable `checkpoint` and the object `this` which are explicitly captured by value (in particular, `p` and `persons_` are implicitly captured by reference here). The second Boost.ScopeExit declaration instead captures all variables in scope by value but `p` which is explicitly captured by reference (in particular, `checkpoint`, `prev_id`, and `this` are implicitly captured by value here).

Note that the `BOOST_SCOPE_EXIT_ALL` macro follows the C++11 lambda function syntax which is unfortunately different from the `BOOST_SCOPE_EXIT` macro syntax. In particular:

1. The `BOOST_SCOPE_EXIT_ALL` macro cannot capture data members without capturing the object `this` while that is not the case for `BOOST_SCOPE_EXIT`. [5]
2. The `BOOST_SCOPE_EXIT_ALL` macro captures the object in scope using `this` instead of `this_`. [6]
3. The `BOOST_SCOPE_EXIT_ALL` body is terminated by a semicolon `;` instead than by the `BOOST_SCOPE_EXIT_END` macro.

If programmers define the configuration macro `BOOST_SCOPE_EXIT_CONFIG_USE_LAMBDAS` then the `BOOST_SCOPE_EXIT` macro implementation will use C++11 lamda functions and the `BOOST_SCOPE_EXIT` macro will follow the same syntax of `BOOST_SCOPE_EXIT_ALL` macro, which is the C++11 lambda function syntax. However, `BOOST_SCOPE_EXIT` will no longer be backward compatible and older code using `BOOST_SCOPE_EXIT` might no longer compile (if data members were explicitly captured).

[5] At present, there seems to be some discussion to allow C++11 lambda functions to capture data members without capturing the object `this`. If the C++11 standard were changed to allow this, the `BOOST_SCOPE_EXIT_ALL` macro syntax could be extended to be a superset of the `BOOST_SCOPE_EXIT` macro while keeping full backward compatibility.

[6] On compilers that support the use of the `typename` outside templates as allowed by the C++11 standard, `BOOST_SCOPE_EXIT_ALL` can use both `this` and `this_` to capture the object in scope (notably, this is not the case for the MSVC 10.0 compiler).

Template Workaround (GCC)

Various versions of the GCC compiler do not compile `BOOST_SCOPE_EXIT` inside templates (see the Reference section for more information). As a workaround, `BOOST_SCOPE_EXIT_TPL` should be used instead of `BOOST_SCOPE_EXIT` in these cases. [7] The `BOOST_SCOPE_EXIT_TPL` macro has the exact same syntax of `BOOST_SCOPE_EXIT`. For example (see also `world_tpl.cpp`):

```
template<typename Person>
void world<Person>::add_person(Person const& a_person) {
    bool commit = false;
    persons_.push_back(a_person);

    BOOST_SCOPE_EXIT_TPL(&commit, this_) { // Use `_TPL` postfix.
        if(!commit) this_->persons_.pop_back();
    } BOOST_SCOPE_EXIT_END

    // ...

    commit = true;
}
```

It is recommended to always use `BOOST_SCOPE_EXIT_TPL` within templates so to maximize portability among different compilers.

Same Line Expansions

In general, it is not possible to expand the `BOOST_SCOPE_EXIT`, `BOOST_SCOPE_EXIT_TPL`, `BOOST_SCOPE_EXIT_END`, and `BOOST_SCOPE_EXIT_ALL` macros multiple times on the same line. [8]

Therefore, this library provides additional macros `BOOST_SCOPE_EXIT_ID`, `BOOST_SCOPE_EXIT_ID_TPL`, `BOOST_SCOPE_EXIT_END_ID`, and `BOOST_SCOPE_EXIT_ALL_ID` which can be expanded multiple times on the same line as long as programmers specify a unique identifiers as the macros' first parameters. The unique identifier can be any token (not just numeric) that can be concatenated by the C++ preprocessor (e.g., `scope_exit_number_1_at_line_123`). [9]

The `BOOST_SCOPE_EXIT_ID`, `BOOST_SCOPE_EXIT_ID_TPL`, and `BOOST_SCOPE_EXIT_ALL_ID` macros accept a capture list using the exact same syntax as `BOOST_SCOPE_EXIT` and `BOOST_SCOPE_EXIT_ALL` respectively. For example (see also `same_line.cpp`):

[7] **Rationale.** GCC versions compliant with C++11 do not present this issue and given that `BOOST_SCOPE_EXIT_ALL` is only available on C++11 compilers, there is no need for a `BOOST_SCOPE_EXIT_ALL_TPL` macro.

[8] **Rationale.** The library macros internally use `__LINE__` to generate unique identifiers. Therefore, if the same macro is expanded more than on time on the same line, the generated identifiers will no longer be unique and the code will not compile. (This restriction does not apply to MSVC and other compilers that provide the non-standard `__COUNTER__` macro.)

[9] Because there are restrictions on the set of tokens that the C++ preprocessor can concatenate and because not all compilers correctly implement these restrictions, it is in general recommended to specify unique identifiers as a combination of alphanumeric tokens.

```
#define SCOPE_EXIT_INC_DEC(variable, offset) \
    BOOST_SCOPE_EXIT_ID(BOOST_PP_CAT(inc, __LINE__), /* unique ID */ \
            &variable, offset) { \
        variable += offset; \
    } BOOST_SCOPE_EXIT_END_ID(BOOST_PP_CAT(inc, __LINE__)) \
    \
    BOOST_SCOPE_EXIT_ID(BOOST_PP_CAT(dec, __LINE__), \
            &variable, offset) { \
        variable -= offset; \
    } BOOST_SCOPE_EXIT_END_ID(BOOST_PP_CAT(dec, __LINE__))

#define SCOPE_EXIT_INC_DEC_TPL(variable, offset) \
    BOOST_SCOPE_EXIT_ID_TPL(BOOST_PP_CAT(inc, __LINE__), \
            &variable, offset) { \
        variable += offset; \
    } BOOST_SCOPE_EXIT_END_ID(BOOST_PP_CAT(inc, __LINE__)) \
    \
    BOOST_SCOPE_EXIT_ID_TPL(BOOST_PP_CAT(dec, __LINE__), \
            &variable, offset) { \
        variable -= offset; \
    } BOOST_SCOPE_EXIT_END_ID(BOOST_PP_CAT(dec, __LINE__))

#define SCOPE_EXIT_ALL_INC_DEC(variable, offset) \
    BOOST_SCOPE_EXIT_ALL_ID(BOOST_PP_CAT(inc, __LINE__), \
            =, &variable) { \
        variable += offset; \
    }; \
    BOOST_SCOPE_EXIT_ALL_ID(BOOST_PP_CAT(dec, __LINE__), \
            =, &variable) { \
        variable -= offset; \
    };

template<typename T>
void f(T& x, T& delta) {
    SCOPE_EXIT_INC_DEC_TPL(x, delta) // Multiple scope exits on same line.
    BOOST_TEST(x == 0);
}

int main(void) {
    int x = 0, delta = 10;

    {
        SCOPE_EXIT_INC_DEC(x, delta) // Multiple scope exits on same line.
    }
    BOOST_TEST(x == 0);

    f(x, delta);

#ifndef BOOST_NO_CXX11_LAMBDAS
    {
        SCOPE_EXIT_ALL_INC_DEC(x, delta) // Multiple scope exits on same line.
    }
    BOOST_TEST(x == 0);
#endif // LAMBDAS

    return boost::report_errors();
}
```

As shown by the example above, the `BOOST_SCOPE_EXIT_ID`, `BOOST_SCOPE_EXIT_ID_TPL`, `BOOST_SCOPE_EXIT_END_ID`, and `BOOST_SCOPE_EXIT_ALL_ID` macros are especially useful when it is necessary to invoke them multiple times within user-defined macros (because the C++ preprocessor expands all nested macros on the same line).

Annex: Alternatives

This section presents some alternative and related work to Boost.ScopeExit.

Try-Catch

This is an example of using a badly designed `file` class. An instance of `file` does not close the file in its destructor, a programmer is expected to call the `close` member function explicitly. For example (see also `try_catch.cpp`):

```
file passwd;
try {
    passwd.open("/etc/passwd");
    // ...
    passwd.close();
} catch(...) {
    std::clog << "could not get user info" << std::endl;
    if(passwd.is_open()) passwd.close();
    throw;
}
```

Note the following issues with this approach:

1. The `passwd` object is defined outside of the `try` block because this object is required inside the `catch` block to close the file.
2. The `passwd` object is not fully constructed until after the `open` member function returns.
3. If opening throws, the `passwd.close()` should not be called, hence the call to `passwd.is_open()`.

The Boost.ScopeExit approach does not have any of these issues. For example (see also `try_catch.cpp`):

```
try {
    file passwd("/etc/passwd");
    BOOST_SCOPE_EXIT(&passwd) {
        passwd.close();
    } BOOST_SCOPE_EXIT_END
} catch(...) {
    std::clog << "could not get user info" << std::endl;
    throw;
}
```

RAII

RAII is absolutely perfect for the `file` class introduced above. Use of a properly designed `file` class would look like:

```
try {
    file passwd("/etc/passwd");
    // ...
} catch(...) {
    std::clog << "could not get user info" << std::endl;
    throw;
}
```

However, using RAII to build up a strong guarantee could introduce a lot of non-reusable RAII types. For example:

```
persons_.push_back(a_person);
pop_back_if_not_commit pop_back_if_not_commit_guard(commit, persons_);
```

The `pop_back_if_not_commit` class is either defined out of the scope or as a local class:

```
class pop_back_if_not_commit {
    bool commit_;
    std::vector<person>& vec_;
    // ...
    ~pop_back_if_not_commit() {
        if(!commit_) vec_.pop_back();
    }
};
```

In some cases strong guarantee can be accomplished with standard utilities:

```
std::auto_ptr<Person> superman_ptr(new superman());
persons_.push_back(superman_ptr.get());
superman_ptr.release(); // persons_ successfully took ownership
```

Or with specialized containers such as Boost.PointerContainer or Boost.Multi-Index.

Scope Guards

Imagine that a new currency rate is introduced before performing a transaction (see also []):

```
bool commit = false;
std::string currency("EUR");
double rate = 1.3326;
std::map<std::string, double> rates;
bool currency_rate_inserted =
        rates.insert(std::make_pair(currency, rate)).second;
// Transaction...
```

If the transaction does not complete, the currency must be erased from `rates`. This can be done with ScopeGuard and Boost.Lambda (or Boost.Phoenix):

```
using namespace boost::lambda;

ON_BLOCK_EXIT(
    if_(currency_rate_inserted && !_1) [
        bind(
            static_cast<
                std::map<std::string, double>::size_type
                (std::map<std::string, double>::*)(std::string const&)
            >(&std::map<std::string, double>::erase)
          , &rates
          , currency
        )
    ]
  , boost::cref(commit)
  );

// ...

commit = true;
```

Note the following issues with this approach:

1. Boost.Lambda expressions are hard to write correctly (e.g., overloaded functions must be explicitly casted, as demonstrated in the example above).

2. The condition in the `if_` expression refers to `commit` variable indirectly through the `_1` placeholder reducing readability.

3. Setting a breakpoint inside `if_[...]` requires in-depth knowledge of Boost.Lambda and debugging techniques.

This code will look much better with C++11 lambdas:

```
ON_BLOCK_EXIT(
    [currency_rate_inserted, &commit, &rates, &currency]() {
        if(currency_rate_inserted && !commit) rates.erase(currency);
    }
);

// ...

commit = true;
```

With Boost.ScopeExit we can simply do the following (see also `scope_guard.cpp`):

```
BOOST_SCOPE_EXIT(currency_rate_inserted, &commit, &rates, &currency) {
    if(currency_rate_inserted && !commit) rates.erase(currency);
} BOOST_SCOPE_EXIT_END

// ...

commit = true;
```

The D Programming Language

Boost.ScopeExit is similar to scope(exit) feature built into the D programming language.

A curious reader may notice that the library does not implement `scope(success)` and `scope(failure)` of the D language. Unfortunately, these are not possible in C++ because failure or success conditions cannot be determined by calling `std::uncaught_exception` (see Guru of the Week #47 for details about `std::uncaught_exception` and if it has any good use at all). However, this is not a big problem because these two D's constructs can be expressed in terms of scope(exit) and a `bool commit` variable (similarly to some examples presented in the Tutorial section).

C++11 Lambdas

Using C++11 lambdas, it is relatively easy to implement the Boost.ScopeExit construct. For example (see also `world_cxx11_lambda.cpp`):

```
#include <functional>

struct scope_exit {
    scope_exit(std::function<void (void)> f) : f_(f) {}
    ~scope_exit(void) { f_(); }
private:
    std::function<void (void)> f_;
};

void world::add_person(person const& a_person) {
    bool commit = false;

    persons_.push_back(a_person);
    scope_exit on_exit1([&commit, this](void) { // Use C++11 lambda.
        if(!commit) persons_.pop_back(); // `persons_` via captured `this`.
    });

    // ...

    commit = true;
}
```

However, this library allows to program the Boost.ScopeExit construct in a way that is portable between C++03 and C++11 compilers.

Annex: No Variadic Macros

This section presents an alternative syntax for compilers without variadic macro support.

Sequence Syntax

Most modern compilers support variadic macros (notably, these include GCC, MSVC, and all C++11 compilers). [10] However, in the rare case that programmers need to use this library on a complier without variaidc macros, this library also allows to specify the capture list using a Boost.Preprocessor sequence where tokens are separated by round parenthesis `()`:

```
(capture1) (capture2) ... // All compilers.
```

Instead of the comma-separated list that we have seen so far which requires variadic macros:

```
capture1, capture2, ... // Only compilers with variadic macros.
```

For example, the following syntax is accepted on all compilers with and without variadic macros (see also `world_seq.cpp` and `world.cpp`):

Boost.Preprocessor Sequence (All Compilers)

```
void world::add_person(person const& a_person) {
    bool commit = false;

    persons_.push_back(a_person);           // (1) direct action
    // Following block is executed when the enclosing scope exits.
    BOOST_SCOPE_EXIT( (&commit) (&persons_) ) {
        if(!commit) persons_.pop_back();    // (2) rollback action
    } BOOST_SCOPE_EXIT_END

    // ...                                  // (3) other operations

    commit = true;                          // (4) disable rollback actions
}
```

Comma-Separated List (Variadic Macros Only)

```
void world::add_person(person const& a_person) {
    bool commit = false;

    persons_.push_back(a_person);           // (1) direct action
    // Following block is executed when the enclosing scope exits.
    BOOST_SCOPE_EXIT(&commit, &persons_) {
        if(!commit) persons_.pop_back();    // (2) rollback action
    } BOOST_SCOPE_EXIT_END

    // ...                                  // (3) other operations

    commit = true;                          // (4) disable rollback actions
}
```

Note how the same macros accept both syntaxes on compilers with variadic macros and only the Boost.Preprocessor sequence syntax on compilers without variadic macros. Older versions of this library used to only support the Boost.Preprocessor sequence syntax so this syntax is supported also for backward compatibility. However, in the current version of this library and on compilers with variadic macros, the comma-separated syntax is preferred because it is more readable.

Finally, an empty capture list is always specified using `void` on compilers with and without variaidc macros (see also `world_void.cpp`):

[10] A C++ compiler does not support variadic macros if the Boost.Config macro `BOOST_NO_CXX11_VARIADIC_MACROS` is defined for that compiler.

```
struct world_t {
    std::vector<person> persons;
    bool commit;
} world; // Global variable.

void add_person(person const& a_person) {
    world.commit = false;
    world.persons.push_back(a_person);

    BOOST_SCOPE_EXIT(void) { // No captures.
        if(!world.commit) world.persons.pop_back();
    } BOOST_SCOPE_EXIT_END

    // ...

    world.commit = true;
}
```

Examples

For reference, the following is a list of most of the examples presented in this documentation reprogrammed using the Boost.Preprocessor sequence syntax instead of comma-separated lists (in alphabetic order):

Files
same_line_seq.cpp
scope_guard_seq.cpp
try_catch_seq.cpp
world_checkpoint_all_seq.cpp
world_checkpoint_seq.cpp
world_this_seq.cpp
world_tpl_seq.cpp

Reference

Acknowledgements

Alexander Nasonov is the original library author.

Lorenzo Caminiti added variadic macro support, capture of the object `this_`, empty captures using `void`, and `BOOST_SCOPE_EXIT_ALL`.

Thanks to the following people (in chronological order):

Maxim Yegorushkin for sharing code where he used a local struct to clean up resources;

Andrei Alexandrescu for pointing out the scope(exit) construct of the D programming language;

Pavel Vozenilek and Maxim Yanchenko for reviews of early drafts of the library;

Steven Watanabe for his valuable ideas;

Jody Hagins for good comments that helped to significantly improve the documentation;

Richard Webb for testing the library on MSVC compiler;

Adam Butcher for a workaround to error C2355 when deducing the type of `this` on some MSVC versions.