# Boost.ScopeExit

Alexander Nasonov



## Table of Contents



# Introduction

Nowadays, every C++ developer is familiar with RAII technique. It binds resource acquisition and release to initialization and destruction of a variable that holds the resource. But there are times when writing a special class for such variable is not worth the effort.

This is when ScopeExit macro comes into play. You put resource acquisition directly in your code and next to it you write a code that releases the resource.

Read Tutorial to find out how to write programs with ScopeExit or jump straight to the Reference section.

# Tutorial

Imagine that you want to make many modifications to data members of the `World` class in the `World::addPerson` function. You start with adding a new `Person` object to a vector of persons:

```
void World::addPerson(Person const& person) {
    bool commit = false;
    m_persons.push_back(person);  // (1) direct action
```

Some operation 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, last added person must be deleted from `m_persons` when the function throws. All you need is to define a delayed action (release of a resource) right after the direct action (resource acquisition):

```
void World::addPerson(Person const& aPerson) {
    bool commit = false;
    m_persons.push_back(aPerson);  // (1) direct action
    BOOST_SCOPE_EXIT( (&commit)(&m_persons) )
    {
        if(!commit)
            m_persons.pop_back(); // (2) rollback action
    } BOOST_SCOPE_EXIT_END

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

    commit = true;                // (4) turn all rollback actions into no-op
}
```

The block below point `(1)` is a ScopeExit declaration. Unlike point `(1)`, an execution of the 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.

The ScopeExit declaration starts with `BOOST_SCOPE_EXIT` macro invocation which accepts Boost.Preprocessor sequence of captured variables. If a capture starts with the ampersand sign `&`, a reference to the captured variable will be available inside the ScopeExit body; otherwise, a copy of the variable will be made after the point `(1)` and only the copy will be available inside the body.

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

Consider a more complex case where `World::addPerson` can save intermediate states at some points and roll back to the last saved state. You can use `Person::m_evolution` to store a version of changes and increment it to cancel all rollback actions associated with those changes.

If you pass a current value of `m_evolution` stored in the `checkpoint` variable by value, it will remain unchanged until the end of aa scope and you can compare it with the final value of the `m_evolution`. If the latter wasn't incremented since you saved it, the rollback action inside the block should be executed:

```
void World::addPerson(Person const& aPerson) {
    m_persons.push_back(aPerson);

    // This block must be no-throw
    Person& person = m_persons.back();
    Person::evolution_t checkpoint = person.m_evolution;

    BOOST_SCOPE_EXIT( (checkpoint)(&person)(&m_persons) )
    {
        if(checkpoint == person.m_evolution)
            m_persons.pop_back();
    } BOOST_SCOPE_EXIT_END

    // ...

    checkpoint = ++person.m_evolution;

    // Assign new id to the person
    World::id_t const prev_id = person.m_id;
    person.m_id = m_next_id++;
    BOOST_SCOPE_EXIT( (checkpoint)(&person)(&m_next_id)(prev_id) )
    {
        if(checkpoint == person.m_evolution) {
            m_next_id = person.m_id;
            person.m_id = prev_id;
        }
    } BOOST_SCOPE_EXIT_END

    // ...

    checkpoint = ++person.m_evolution;
}
```

Full code listing can be found in world.cpp.

# Alternatives

## try-catch

This is an example of using a badly designed `File` class. An instance of `File` doesn't close a file in a destructor, a programmer is expected to call the `close` member function explicitly.

```
File passwd;
try {
    passwd.open("/etc/passwd");
    // ...
    passwd.close();
}
catch(...) {
    log("could not get user info");
    if(passwd.is_open())
        passwd.close();
    throw;
}
```

Note the following:

- the `passwd` object is defined outside of the `try` block because this object is required inside the `catch` block to close the file,
- the `passwd` object is not fully constructed until after the `open` member function returns, and

- if opening throws, the `passwd.close()` should not be called, hence the call to `passwd.is_open()`.

ScopeExit doesn't have any of these problems:

```
try {
    File passwd("/etc/passwd");
    BOOST_SCOPE_EXIT( (&passwd) ) {
        passwd.close();
    } BOOST_SCOPE_EXIT_END
    // ...
}
catch(...) {
    log("could not get user info");
    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(...) {
    log("could not get user info");
    throw;
}
```

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

```
m_persons.push_back(person);
pop_back_if_not_commit pop_back_if_not_commit_guard(commit, m_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 m_commit;
    std::vector<Person>& m_vec;
    // ...
    ~pop_back_if_not_commit() {
        if(!m_commit)
            m_vec.pop_back();
    }
};
```

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

```
std::auto_ptr<Person> spSuperMan(new Superman);
m_persons.push_back(spSuperMan.get());
spSuperMan.release(); // m_persons successfully took ownership.
```

or with specialized containers such as Boost Pointer Container Library or Boost Multi-Index Containers Library.

# ScopeGuard

Imagine that you add a new currency rate:

```
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;
```

and then continue a transaction. If it cannot be completed, you erase the currency from `rates`. This is how you can do this with ScopeGuard and Boost.Lambda:

```
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 that

- Boost.lambda expressions are hard to write correctly, for example, overloaded function must be explicitly casted, as demonstrated in this example,
- condition in `if_` expression refers to `commit` variable indirectly through the `_1` placeholder,
- setting a breakpoint inside `if_[ ... ]` requires in-depth knowledge of Boost.Lambda and debugging techniques.

This code will look much better with native lambda expressions proposed for C++0x:

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

With ScopeExit you can simply do

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

// ...

commit = true;
```

## C++0x

In future releases ScopeExit will take advantages of C++0x features.

- Passing capture list as Boost.Preprocessor sequence will be replaced with a traditional macro invocation style:

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

- `BOOST_SCOPE_EXIT_END` will be replaced with a semicolon:

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

- Users will be able to capture local variables implicitly with lambda capture defaults `&` and `=`:

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

- It will be possible to capture `this` pointer.

## The D Programming Language

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

A curious reader may notice that the library doesn't implement `scope(success)` and `scope(failure)` of the D language. Unfortunately, it's not possible in C++ because failure or success condition cannot be determined by calling `std::uncaught_exception`. It's not a big problem, though. These two constructs can be expressed in terms of scope(exit) and a `bool commit` variable as explained in Tutorial. Refer to Guru of the Week #47 for more details about `std::uncaught_exception` and if it has any good use at all.

# Supported Compilers

The 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 ScopeExit is included by multiple translation units.
- GCC 3.3 can't compile ScopeExit inside a template. See this thread for more details.

The author tested the library on GCC 3.3, 3.4, 4.1, 4.2 and Intel 10.1.

# Configuration

Normally, no configuration is required for the library but note that the library depends on Boost.Typeof and you may want to configure or enforce typeof emulation.

# Reference

## BOOST_SCOPE_EXIT

A ScopeExit declaration has the following synopsis:

```
#include <boost/scope_exit.hpp>

BOOST_SCOPE_EXIT ( scope-exit-capture-list )
    function-body
BOOST_SCOPE_EXIT_END
```

where

```
scope-exit-capture-list:
    ( scope-exit-capture )
    scope-exit-capture-list ( scope-exit-capture )

scope-exit-capture:
    identifier
    &identifier
```

The ScopeExit declaration schedules an execution of `scope-exit-body` at the end of the current scope. The `scope-exit-body` statements are executed in the reverse order of ScopeExit declarations in the given scope. The scope must be local.

Each `identifier` in `scope-exit-capture-list` must be a valid name in enclosing scope and it must appear exactly once in the list. If a `scope-exit-capture` starts with the ampersand sign `&`, the corresponding `identifier` will be available inside `scope-exit-body`; otherwise, a copy of it will be made at the point of ScopeExit declaration and that copy will be available inside `scope-exit-body`. In the latter case, the `idenitifer` must be `CopyConstructible`.

Only identifiers listed in `scope-exit-capture-list`, static variables, `extern` variables and functions, and enumerations from the enclosing scope can be used inside the `scope-exit-body`.

**Note**

`this` pointer is not an identifier and cannot be passed to `scope-exit-capture-list`.

The ScopeExit uses Boost.Typeof to determine types of `scope-exit-capture-list` elements. In order to compile code in typeof emulation mode, all types should be registered with BOOST_TYPEOF_REGISTER_TYPE or BOOST_TYPEOF_REGISTER_TEMPLATE macros, or appropriate Boost.Typeof headers should be included.

## BOOST_SCOPE_EXIT_TPL

This macro is a workaround for various versions of gcc. These compilers don't compile ScopeExit declaration inside function templates. As a workaround, the `_TPL` suffix should be appended to `BOOST_SCOPE_EXIT`.

The problem boils down to the following code:

```
template<class T> void foo(T const& t) {
    int i = 0;
    struct Local {
        typedef __typeof__(i) typeof_i;
        typedef __typeof__(t) typeof_t;
    };
    typedef Local::typeof_i i_type;
    typedef Local::typeof_t t_type;
}

int main() { foo(0); }
```

This can be fixed by adding `typename` in front of `Local::typeof_i` and `Local::typeof_t`.

See also GCC bug 37920.

**Note**

Although `BOOST_SCOPE_EXIT_TPL` has the same suffix as the `BOOST_TYPEOF_TPL`, it doesn't follow a convention of the Boost.Typeof.

# Acknowledge

(in chronological order)

Maxim Yegorushkin for sending me a code where he used a local struct to clean up resources.

Andrei Alexandrescu for pointing me to 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.