Table of Contents

1 Introduction

1.1 What

Simply stated, this library enables the use of the Pythonesque `with`-statement construct in C++. More broadly, this library seeks to formalize a generalized version of a subset of the *RAII (Resource Acquisition is Initialization)* idiom that, for the lack of preexisting terms, I shall refer here to as the *STEP (Scope Triggered Event Processing)* idiom. The *STEP* idiom is the commonly occurring use case where for a given block of code some user specified action(s) must execute on entry and on exit, and the failure to do so will result in some deterministic behavior. Some examples of such use cases are transactional programs where a series of steps must all be executed successfully or else all rolled back, and GUI programs where it is desired to suspend animation before modifying GUI elements and resume animation thereafter.

1.2 Why

1.2.1 Why Formalize?

The constructs identified by the *STEP* idiom usually entail boilerplate code before and after some scope. Not only is it repetitive and tedious to reproduce the boilerplate code by hand, it is also error prone since it's easy to get it almost, but not quite, right through copy-and-past errors. Formalization takes care of the boilerplate code, allowing the programmer to focus on the task at hand. And when done correctly, formalization makes for more readable and maintainable code by clearly conveying and enforcing the intent of the formalized idiom.

1.3 How

The macros `WG_LCL_CONTEXT`/`WGLCLCONTEXT_TPL`/`WG_LCLCONTEXT_END`*N* are used to implement the *STEP* idiom in this library. Associated with the use of these macros are scope managers. Loosely, scope managers are any objects with a no-argument method named `enter`, and one-argument method named `exit`. For the scope defined in between the macro invocations of `WG_LCL_CONTEXT`/`WGLCLCONTEXT_TPL` and `WG_LCLCONTEXT_END`*N*, it is guaranteed that the `enter` method of any associated scope managers will be called in the order that the scope managers were declared in the "opening macro", and that the `exit` method of those same scope managers will be called in the reverse of the said order. If such a scope is exited "prematurely", then the said `exit` methods will be called with the boolean value `false`, else they will be called with the boolean value `true`. In addition, it is also possible to create on-the-fly scope managers using variables bound from the enclosing scope. Such scope managers are called adhoc scope managers whereas the other ones are called extant scope managers.

The following examples should give a quick overview of how these macros can be used.

1.3.1 Extant

```
#include <WG/Local/LclContext.hh>

template <typename StreamOrBuf>
struct FileMngr
{
  explicit FileMngr(char const * pFilepath)
  : m_Filepath(pFilepath),
    m_StreamOrBuf()
  {}

  StreamOrBuf & enter()
  {
    m_StreamOrBuf.open(m_Filepath.c_str());
    return m_StreamOrBuf;
  }

  void exit(bool const scope_completed)
  {
    (void)scope_completed;
    m_StreamOrBuf.close();
  }

private:
  std::string m_Filepath;
  StreamOrBuf m_StreamOrBuf;
};

void write()
{
  WG_LCLCONTEXT(
    with(FileMngr<std::ofstream>("tmp.bin")) enter_as(ref filestrm) )
  // Scope1
  {
    // Write to filestrm ...
  }
  WG_LCLCONTEXT_END1
}
```

In the function `write`, an anonymous scope manager is associated with the opening macro and upon entry into `Scope1` its `enter` method is invoked, the result of that method invocation is then captured by reference and made available in Scope1 via the variable `filestrm`. Upon exit from Scope1, for any reason whatsoever, the `exit` method of that anonymous scope manager will then be invoked.

1.3.2 Adhoc

```
#include <WG/Local/LclContext.hh>

Transaction transaction;

WG_LCLCONTEXT(
  with_adhoc (ref transaction)
    on_enter(transaction.start();)
    on_exit(
      if(! scope_completed) {transaction.rollback();}
      else { transaction.commit(); }) )
// Scope1
```

```
{
  // record transactions.
}
WG_LCLCONTEXT_END1
```

Above, an anonymous scope manager object is created on-the-fly with a reference to the previously declared `transaction` object. Upon entry into `Scope1`, said scope manager will execute the snippet of code specified in the `on_enter` named parameter, and upon exit from that scope, for any reason whatsoever, it will execute the code snippet specified in the `on_exit` named parameter. Note that the variable `scope_completed` is available to determine whether the associated STEPed scope exited prematurely or not.

2 Tutorial

2.1 A Note on Syntax

2.1.1 Named Macro Parameters

This library uses named parameters to pass arguments to the macros that implement it. It is this author's opinion that this makes such macros clearer to understand and easier to use. Three types of named macro parameters are used, those that expect value expressions, those that expect bound variable declarations, and those that expect C++ compound statements. Value expressions are C++ expressions that evaluate to some object, bound variable declarations are DSEL variable declarations whose variable identifiers bind to preexisting variables of the same name in the enclosing scope of the said declaration, and C++ compound statements are sequences of C++ statements.

2.1.1.1 Bound Variable Declarations

Bound variable declarations maybe implicitly typed or explicitly typed. Implicitly typed bound variable declarations deduce their type from the identifier they bind to using Boost.Typeof. The use of Boost.Typeof library is not strictly necessary, those that wish to forgo its use can explicitly type their bound variable declarations.

> *Advice:* It is recommended that implicitly typed bound variable declarations be used wherever possible. This relieves the burden of having to manually keep the type of the binder in the named macro parameter argument and the type of its bindee in sync, thus resulting in more maintainable code. Note that a binder/bindee type mismatch will not always result in a compiler error. This is true for the case where there exists an implicit conversion sequence from the bindee to the binder.

2.1.1.2 Explicitly Typed Syntax

The syntax for an explicitly typed variable declaration is:

```
type( non-local-type-specifier )
```

Where the appropriate terms are defined here.

> *Advice:* Note that commas cannot be used in arguments to `type`. (For rationale see Macro Parameter Arguments...)

2.1.1.3 Implicitly Typed Syntax

The syntax for an implicitly typed variable declaration is:

```
const | ref | const ref | empty-token
```

Where the keyword `const` const qualifies the result of Boost.Typeof and the keyword `ref` reference qualifies it.

> *Note:* As a result, implicitly typed variables cannot be named `ref`.

2.1.1.4 Other Named Parameter Types

> *Advice:* For maximal portability with C++03, it is recommended that commas in named parameters other than those of bound variable declarations be enclosed in an extra set of parenthesis. (For rationale see Macro Parameter Arguments...)

2.1.1.5 Non-Variadic Arguments

For those that wish to strictly stay C++03 conformant, it is possible to forgo the use of variadic macros when using this library. In that case named macro parameters with variadic argument lists should have the latter list, including the enclosing parenthesis, replaced by a sequence of tuples where each tuple element represents an argument to the said macro. For example, "`named_param(x, y, z)`" should be replaced by "`named_param (x) (y) (z)`".

2.2 A Note on Examples

All examples in this document will, whenever possible, use implicitly typed, variadic named macro parameter arguments.

2.3 The Implementing Macros

The macros `WG_LCL_CONTEXT`/`WGLCLCONTEXT_TPL` and `WG_LCLCONTEXT_END`*N* start and end a scope that upon entry executes some user specified action and upon exit, for any reason whatsoever, executes some other user specified action. This scope starts after the invocation of `WG_LCL_CONTEXT`/`WGLCLCONTEXT_TPL` and ends before the invocation of `WG_LCLCONTEXT_END`*N*. `WG_LCL_CONTEXT`/`WGLCLCONTEXT_TPL` each is said to be an opening macro, `WG_LCLCONTEXT_END`*N* is said to be a closing macro, and the scope they define in between them is said to be a STEPed scope. The *`N`* in `WG_LCLCONTEXT_END`*N* represents the number of scope managers declared in the opening macro.

`WG_LCL_CONTEXT` and `WG_LCL_CONTEXT_TPL` have the same fundamental functionality and the same syntax, they only differ in that the former is for use in non-template functions and the latter is for use in template functions. Fundamental to the work of these macros are scope managers.

2.4 Scope Managers

Scope managers are central to this implementation of the *STEP* idiom. Each of the opening macros `WG_LCL_CONTEXT`/`WGLCLCONTEXT_TPL` is associated with at least one scope manager. Scope managers are used to trigger user specified code whenever a STEPed scope is entered and exited. They are defined to be any object that has at least one publicly accessible method from each of the following method categories:

2.4.1.1 Entry Methods

```
some-unspecified-type enter();
some-unspecified-type enter() const;
```

2.4.1.2 Exit Methods

```
void exit(bool const scope_completed);
void exit(bool const scope_completed) const;
```

Whenever a STEPed scope is entered the associated scope manager(s) invoke their `enter` method in the order in which they were declared in the opening macro; and whenever a STEPed scope is exited the associated step manager(s) invoke their `exit` method in the reverse order in which they were declared in the opening macro. If the STEPed scope was executed prematurely, that is control was transferred from within the STEPed scope to outside of it due to the execution of a `goto`, `return`, `break`, `continue`, or `throw` statement, then the aforementioned `exit` method(s) will be called with the boolean value `false`, else they will be called with the boolean value `true`.

Note that `enter` methods are allowed to throw an exception. This means that if one does actually throw an exception, then the only `exit` methods that will subsequently be called are:

1. the one for the throwing `enter` method,
2. those associated with any previously invoked `enter` methods from the same opening macro,
3. and those from any opening macro whose STEPed scope the current code may be nested in.

Also note that `exit` methods are allowed to throw an exception. If an `exit` method does throw an exception, then any subsequent `exit` methods that are slated to be invoked will still be invoked.

> *Note:* An `exit` method that does throw will silently consume any active exception that originated from any nested scope of its associated STEPed scope.

2.5 Extant Scope Managers

2.5.1 `with` Parameter

Existing scope managers are any objects accessible within the current scope that meet the definition of a scope manager. They are specified in the `with` named parameter of their opening macro. The `with` statement captures by the reference (lvalues), or by value (rvalues), the result of any C++ expression it's given and treats that captured object as a scope manager. (Note that this is all done without resorting to the use of Boost.Typeof.)

> *Advice:* This library is Boost.Move enabled, so it's advised that scope managers also be Boost.Move enabled.

> *Note:* Non-copyable const rvalues may not be used as scope managers. That's because the latter can neither be copied nor moved. Additionally, for portability with C++03 it is recommended that non-copyable scope managers be marked as such. For further information see Noncopyable ...

Example:

```
#include <WG/Local/LclContext.hh>

layout_manager lmngr(widget1, widget2, widget3);
WG_LCLCONTEXT( with(lmngr) )
{
  widget1.x = 450;
  widget1.y = 500;

  widget2.height = 10;
  widget2.width = 20;

  widget3.color = Blue;
}
WG_LCLCONTEXT_END1
```

2.5.2 `enter_as` Parameter

The return value, if any, of an extant scope manager's entry maybe captured via the `enter_as` named macro parameter, else it is ignored. The syntax for this is:

```
enter_as([type-expression] variable-name)
```

Example:

```
#include <WG/Local/LclContext.hh>

WG_LCLCONTEXT( with(filemngr("data.txt")) enter_as(file) )
{
  // Process file obj here ...
}
WG_LCLCONTEXT_END1
```

2.6 Adhoc Scope Managers

There are two type of adhoc scope managers, one designated by the `with_adhoc` named macro parameter and the other designated by the `with_raii` macro named parameter.

2.6.1 `with_adhoc` Parameter

Adhoc scope managers are scope managers built on the fly using whatever user specified variable that is visible in the current scope. They are specified in the `with_adhoc` named parameter of the opening macro. The arguments to `with_adhoc` bind those variables with the same name in the enclosing scope to the scope manager that will be built.

2.6.1.1 `this_` Argument

The identifier `this_`, if used as an argument to `with_adhoc` and implicitly typed, binds the `this` variable of the enclosing scope to its associated scope manager. If it is `const` qualified, then the `const` qualification will apply to the pointed-to-type (see Rationale).

2.6.1.2 No Arguments

It is possible to use `with_adhoc` with no arguments, in that case its proceeding opening and closing parenthesis must also be omitted.

Following `with_adhoc`, at least one of `one_enter` or `on_exit` named parameters must be specified. These named parameters, collectively referred to as adhoc scope handlers, are said to be associated with the adhoc scope manager identified by the `with_adhoc` named parameter that immediately precedes them.

2.6.1.3 `on_enter` Parameter

If the `on_enter` named macro parameter is specified, then whatever statements specified as its argument will be executed by its associated adhoc scope manager upon entry into the STEPed scope that immediately follows its opening macro. Within this parameter's argument, those variables bound to its associated scope manager are available for use. For convenience, semicolon will always be appended to this parameter's argument.

2.6.1.4 `on_exit` Parameter

If the `on_exit` named macro parameter is specified, then whatever statements specified as its argument will be executed by its associated adhoc scope manager upon exit, for any reason whatsoever, from the STEPed scope that immediately follows its opening macro. Within this parameter's argument, those variables bound to its associated scope manager are available for use. Additionally, the boolean variable `scope_completed` is also available, indicating whether the STEPed scope exited prematurely or not. For convenience, semicolon will always be appended to this parameter's argument.

Example:

```
#include <WG/Local/LclContext.hh>

File file(path);

WG_LCLCONTEXT(
  with_adhoc(ref file) on_enter( file.open(); ) on_exit( file.close(); )
)
{
  // Process file here ...
}
WG_LCLCONTEXT_END1
```

2.6.2 `with_raii` Parameter

Existing RAII objects maybe used by employing the `with_raii` adaptor. `with_raii` takes a single argument that must be an instantiation of a named RAII object.

> *NOTE:* The argument to `with_raii` named macro parameter may not contain any commas since commas themselves act as preprocessor delimiters.

Example:

```
#include <WG/Local/LclContext.hh>

WG_LCLCONTEXT( with_raii(filemngr_t filemngr("../data.txt");) )
{
    //Do something
}
WG_LCLCONTEXT_END1
```

2.7 Nesting STEPed Macros

STEPed macros maybe nested within one another. However, note that a STEPed macro with multiple scope manager arguments is not equivalent to a series of single scope manager argument nested macros. That is, the following:

```
 WG_LCLCONTEXT( with(scpmngr1) with(scpmngr2) with(scpmngr3) )
{
  ...
}
WG_LCLCONTEXTEND3
```

is NOT equivalent to:

```
WG_LCLCONTEXT( with(scpmngr1) )
{
  WG_LCLCONTEXT( with(scpmngr2) )
  {
    WG_LCLCONTEXT( with(scpmngr3) )
    {
      ...
    }
    WG_LCLCONTEXTEND1
  }
  WG_LCLCONTEXTEND1
}
WG_LCLCONTEXTEND1
```

This is because if `scpmngr3.enter()` throws then in the first example above `scpmngr2.exit` and `scpmngr3.exit` will both be called with `scope_completed = true`, but in the second example they will be called with `scope_completed = false`. Please remember that a STEPed scope is defined to start at the end of an opening macro invocation and end at the start of the following non-nested closing macro invocation.

2.8 Same Line Declarations

Multiple STEPED macros may be declared on the same line, for example, as part of a larger macro definition.

3 Reference

3.1 Macros

3.1.1 `WG_LCLCONTEXT(...)` `WG_LCLCONTEXT_TPL(...)` `WG_LCLCONTEXT_END`*N*

These macros start and end a STEPed scope. Their syntax is defined here. `WG_LCLCONTEXT` is for use in non-template functions and `WG_LCLCONTEXT` is for use in template functions. The *N* in `WG_LOCALCONTEXT_END`*N* must be a number that matches the number of scope managers associated with one of its opening macros. These two aforementioned macros, sometimes referred to as opening macros, will accept the following named parameters:

a) `with(scope-manager-expr)`

The `with` named parameter associates the scope manager specified by its argument with its associated macro. Lvalue expression arguments are captured by reference whereas rvalue expression arguments are captured by value. Whenever possible this library will attempt to move `with`'s rvalue arguments before resorting to copying them.

Non-copyable const rvalues may not be used as arguments to `with`. This is because they can neither be copied nor moved.

For portability with C++03, it is recommended that non-copyable scope managers follow these guidelines.

`enter_as([type-expression] variable-name)`

If present, the `enter_as` named parameter must immediately follow a `with` named parameter. This parameter tells its associated macro to capture the return value of its associated scope manager's `enter` method call in the variable `variable-name`. This variable will only be available within this named parameter's associated STEPed scope. If this named parameter is omitted then the return value, if any, of its associated scope manager's `enter` method call will be ignored.

The type of `variable-name` may optionally be omitted, in which Boost.Typeof will be used to deduce its type.

b) `with_adhoc[ ( nlt-bound-var-dcln-list ) ] adhoc-scope-handlers` (V)

`with_adhoc[ nlt-bound-tuple-seq ] adhoc-scope-handlers`

The `with_adhoc` named parameter creates an anonymous scope manager whose `enter` and `exit` methods can be customized by the `on_enter` and `on_exit` named parameters. Arguments to `with_adhoc` bind those variables that have the same name in the enclosing scope to the said scope manager, making those variables accessible to its handler methods. The special identifier `this_`, if implicitly typed, binds the `this` variable (if there is one) of the enclosing scope to the said scope manager. Take note that `const` qualification of the latter will apply to its pointed-to-type (see Rationale).

Note: at least one of `on_enter` or `on_exit` named parameters must follow this named parameter.

c) `on_enter( compound-statement )`

If present, the `on_enter` named parameter must immediately follow a `with_adhoc` named parameter. This named parameter allows the user to customize the `enter` method of the previously created anonymous scope manager. Please note that the return type of this method will always be `void`, so even if the argument to this named parameter returns, it will be ignored. For convenience, semicolon will always be appended to this parameter's argument.

d) `on_exit( compound-statement )`

If present, the `on_exit` named parameter must either follow a `with_adhoc` or a `on_enter` named parameter. This named parameter allows the user to customize the `exit` method of the previously created anonymous scope manager. Please note that the boolean `scope_completed` variable is accessible within this named parameter's argument. Also note that the return type of this method will always be `void`, so even if the argument to this named parameter returns, it will be ignored. For convenience, semicolon will always be appended to this parameter's argument.

Multiple scope managers, both extant and adhoc, maybe specified by an opening macro. These scope managers are captured and/or constructed in the order in which they appeared in their opening macro, and, whenever appropriate, are destructed in the reverse order. Upon entry into a STEPed scope, the `enter` method of the opening macro's scope managers will be executed in the order in which those scope managers were specified, and upon exit from the said scope, for any reason whatsoever, the `exit` method of said scope managers will be executed in the reverse order.

3.1.2 BOOST_NO_EXCEPTIONS

If this macro is defined then exception supported code will not be generated. This means that if an exception is thrown from a STEPed scope, then none of the guarantees concerning premature scope exit hold for any of the associated scope managers.

4 Limitations

4.1 General Limitations

4.1.1 Macro Parameter Arguments Containing Commas

Non-type macro parameter arguments that contain commas should be enclosed in an extra set of parenthesis. That's because commas act as preprocessor delimiters, thus making it impossible to parse such arguments without help from the user.

As for type macro parameter arguments a very simple workaround is to use typedef aliases in their place.

4.1.2 Implicitly Typed Variables

Implicitly typed variables in named macro parameters may not be named "ref".

4.1.3 Noncopyable Const LValue Scope Managers

If a noncopyable const lvalue expression is to be used with the `with` named parameter in C++03, then its type should be marked to indicate that it is noncopyable. There are four ways to do this:

1. derive said type from ::`boost::noncopyable`, or
2. mark said type `BOOST_MOVABLE_BUT_NOT_COPYABLE`, or
3. mark said type's copy constructor "= delete", or
4. specialize `::boost::copy_constructible` for said type.

For maximum portability, it is advised to do one of the above for all noncopyable scope manager types.

5 Grammar

5.1 Special Symbols

a) [...]

   Items enclosed in square brackets denote optional grammar entries.

b) `with with_raii with_adhoc enter_as on_enter on_exit`
   These tokens are to be regarded as non-punctuation terminals.

5.2 EBNF

```
lclcontext-usage ::=
  lclcontext-start-macro ( lclcontext-spec )
  compound-statement
  WG_LCLCONTEXT_END [;]

lclcontext-start-macro ::=
    WG_LCLCONTEXT
  | WG_LCLCONTEXT_TPL

lclcontext-spec ::=
    with-dcln-stmnt [ lclcontext-spec ]
  | with-raii-stmnt [ lclcontext-spec ]
  | with-adhoc-dcln-stmnt [ lclcontext-spec ]

with-dcln-stmnt ::=
  with( scope-manager-expr ) [ enter_as( nlt-type-var-dcln ) ]
```

```
with-raii-stmnt ::=
  with_raii( compound-statement )

with-adhoc-dcln-stmnt ::=
    with_adhoc[ ( nlt-bound-var-dcln-list ) ] adhoc-scope-handlers (V)
  | with_adhoc[ nlt-bound-tuple-seq ] adhoc-scope-handlers

adhoc-scope-handlers ::=
  [on_enter( compound-statement )] [on_exit( compound-statement )]

nlt-bound-var-dcln-list ::=
    nlt-bound-var-dcln
  | nlt-bound-var-dcln-list , nlt-bound-var-dcln

nlt-bound-tuple-seq ::=
    nlt-bound-tuple
  | nlt-bound-tuple-seq nlt-bound-tuple

nlt-bound-tuple ::=
  ( nlt-bound-var-dcln )

nlt-type-var-dcln ::=
    implicit-non-local-type-var-dcln
  | explicit-non-local-type-var-dcln

implicit-non-local-type-var-dcln ::=
  implicit-type-var-dcln

explicit-non-local-type-var-dcln ::=
  explicit-non-local-type var-name

implicit-type-var-dcln ::=
  implicit-type var-name

type-expression ::=
    explicit-non-local-type
  | implicit-type

implicit-type ::= lib-type-qualifiers | empty-token

lib-type-qualifiers ::= const | ref | const ref

explicit-non-local-type ::=
  type( non-local-type-specifier )

(V) Requires a variadic macro supported preprocessor.
value-expression ::= A C++ expression that evaluates to a value.
scope-manager-expr ::= A C++ expression that evaluates to a scope manager.
compound-statement ::= See C++ standard.
var-name ::= A C++ variable name.
non-local-type-specifier ::=
  A type-specifier that specifies a non-local type.
empty-token ::= The token consisting of no characters.
```

6 Rationale

6.1 this_ const Qualification

The `const` qualification of an implicitly typed `this_` bound variable applies to the pointed-to-type of `this_` because its type, and hence the type of `this`, are already `const` qualified. Not only would it be redundant, but it would also be impossible to `const` qualify the pointed-to-type of an implicitly typed `this_` had the constness been applied to the pointer type.

7 Alternatives

7.1 Why not Boost.ScopeExit?

7.1.1 Unenforceable Intent

Using Boost.ScopeExit to formalize the *STEP* idiom makes enforcing the idiom's intent impossible. This has to do with the fact that Boost.ScopeExit deals with scope exit, and not scope entry. Though the STEP idiom can be correctly implemented by placing the scope entry code directly before its Boost.ScopeExit complement, the programmer has to manually ensure that this done in the right order for all such operations. For example, if for the following three objects: `a1`, `a2`, and `a3` the method `suspend_layout` must be called on scope entry and the method `resume_layout` must be called on scope exit, then the programmer must ensure that they are ordered in the following manner:

```
a1.suspend_layout();
BOOST_SCOPE_EXIT(&foo) { a1.resume_layout(); } BOOST_SCOPE_EXIT_END
a2.suspend_layout();
BOOST_SCOPE_EXIT(&foo) { a2.resume_layout(); } BOOST_SCOPE_EXIT_END
a3.suspend_layout();
BOOST_SCOPE_EXIT(&foo) { a3.resume_layout(); } BOOST_SCOPE_EXIT_END

// User code goes here:
```

and not in the following plausible manner:

```
a1.suspend_layout();
a2.suspend_layout();
a3.suspend_layout();
BOOST_SCOPE_EXIT(&foo) { a1.resume_layout(); } BOOST_SCOPE_EXIT_END
BOOST_SCOPE_EXIT(&foo) { a2.resume_layout(); } BOOST_SCOPE_EXIT_END
BOOST_SCOPE_EXIT(&foo) { a3.resume_layout(); } BOOST_SCOPE_EXIT_END

// User code goes here:
```

since in the latter case if `a2.suspend_layout()` throws then `a1` won't resume its drawing after it has been suspended.

7.1.2 Unclear Intent

Boost.ScopeExit does not clearly convey its intent. For which scope does Boost.ScopeExit apply? The answer is implicit, it's the current scope **after** the `BOOST_SCOPE_EXIT_END` macro. In the author's opinion, a first glance at code which uses Boost.ScopeExit conveys that the specified exit-code will be executed upon exit from anywhere in the current scope and not just for the *portion* of the current scope after Boost.ScopeExit's ending macro. Additionally, this author argues that one has to become acclimated to the latter fact in order to comfortably use Boost.ScopeExit.

7.1.3 Feature Deficiency

With Boost.ScopeExit it is not possible to automatically determine whether its scope was exited prematurely or not. This means that writing transactional code using Boost.ScopeExit requires the programmer to manually indicate the end of scope. Granted, with this library the programmer is also required to put `WG_LCLCONTEXT_ENDN` at the end of scope, however, the difference is that in the former if the manual indicator was forgotten it may or may not result in a runtime error whereas in the latter forgetting to use `WG_LCLCONTEXT_ENDN` will always result in a compile time error.

8 Implementation

The following is an outline of what a STEPed code might generate. The purpose of this is to inform the curious reader and aid library maintainers, but it is not a guarantee of what the implementation will generate.

8.1 Sample STEPed Macro Use

```
#include <WG/Local/LclContext.hh>

WG_LCLCONTEXT(
  with(FileMngr<std::ofstream>("tmp.bin")) enter_as(ref filestrm)
  with_adhoc (ref transaction)
    on_enter(transaction.start();)
    on_exit(
      if(! scope_completed) {transaction.rollback();}
      else { transaction.commit(); })
)
{
  // User code goes here.
}
WG_LCLCONTEXT_END2
```

8.2 Sample STEPed Macro Expansion

```
{
  struct wgXXXlclcontextXXXtypealiaserXXXenteredas_dclns
  {
    typedef
      boost::add_reference
      <
        BOOST_TYPEOF((FileMngr<std::ofstream>("tmp.bin")) . enter())
      >::type
        type0 ;
  };
```

```
    bool wgXXXlclcontextXXXdid_scope_complete = false;
    bool wgXXXlclcontextXXXautosimflag = false;
    (void)wgXXXlclcontextXXXautosimflag ;

    ::wg::lclcontext::detail::extant_scopemngr_proxy_t
      wgXXXlclcontextXXXscopemngrXXX0 =
        ::wg::lclcontext::detail::make_extant_scopemngr_proxy(
          WG_AUTOSIMULATOR_DETAIL_AUTOANY_EXPR_CAPTURE(
            FileMngr<std::ofstream>("tmp.bin"),
            wgXXXlclcontextXXXautosimflag) ) ;
  #ifndef BOOST_NO_EXCEPTIONS
    try
    {
  #endif
      wgXXXlclcontextXXXtypealiaserXXXenteredas_dclns::type0 filestrm =
        ::wg::lclcontext::detail::extant_scopemngr_proxy_downcast(
          wgXXXlclcontextXXXscopemngrXXX0,
          WG_AUTOSIMULATOR_DETAIL_AUTOANY_AUTOANYIMPL_DEDUCEDPTRTYPE(
            FileMngr<std::ofstream>("tmp.bin")) )
        . enter<wgXXXlclcontextXXXtypealiaserXXXenteredas_dclns::type0>();

      struct wgXXXlclclassXXXtypealiaserXXXwgXXXlclcontextXXXadhoc_scopemngr_typeXXX0XXX17
      {
        typedef
          ::boost::add_reference< BOOST_TYPEOF(transaction)) >::type
            memlike0 ;
      };

      class wgXXXlclcontextXXXadhoc_scopemngr_typeXXX0 :
        private ::wg::lclclass::detail::initializer
      {
      private:

wgXXXlclclassXXXtypealiaserXXXwgXXXlclcontextXXXadhoc_scopemngr_typeXXX0XXX17::memlike0
transaction ;

        bool m_didcallexit ;

      public:
        explicit wgXXXlclcontextXXXadhoc_scopemngr_typeXXX0 (
          ::boost::call_traits <
wgXXXlclclassXXXtypealiaserXXXwgXXXlclcontextXXXadhoc_scopemngr_typeXXX0XXX17::memlike0
>::param_type param0 )
        : transaction ( param0 ),
          m_didcallexit( false )
        { this->init(); }

        private:
        public:

        void enter()
        { transaction.start(); ; }

        void exit(bool const scope_completed)
        {
          m_didcallexit = true; (void)scope_completed;
          if(! scope_completed) {transaction.rollback();}
          else { transaction.commit(); } ;
```

```
      }

      ~ wgXXXlclcontextXXXadhoc_scopemngr_typeXXX0 ()
      {
        if( ! m_didcallexit ) { this->exit(false); }
      }
    }; ;

    wgXXXlclcontextXXXadhoc_scopemngr_typeXXX0
      wgXXXlclcontextXXXscopemngrXXX1 ( transaction ) ;

#ifndef BOOST_NO_EXCEPTIONS
    try
    {
#endif
      wgXXXlclcontextXXXscopemngrXXX1 . enter() ;
      {
        // User code goes here.
      }
      wgXXXlclcontextXXXdid_scope_complete = true;
#ifndef BOOST_NO_EXCEPTIONS
    }
    catch(...)
    {
      wgXXXlclcontextXXXscopemngrXXX1 . exit(
        wgXXXlclcontextXXXdid_scope_complete );
      throw;
    }
#endif
    wgXXXlclcontextXXXscopemngrXXX1 . exit(
      wgXXXlclcontextXXXdid_scope_complete );
#ifndef BOOST_NO_EXCEPTIONS
  }
  catch(...)
  {
    wgXXXlclcontextXXXscopemngrXXX0 . exit(
      wgXXXlclcontextXXXdid_scope_complete );
    throw;
  }
#endif
  wgXXXlclcontextXXXscopemngrXXX0 . exit(
    wgXXXlclcontextXXXdid_scope_complete );
}
```

Implementation notes:

1. Extant scope managers are wrapped in proxies of type `extant_scopemngr_proxy_t` so that in the proxy's destructor it will be possible to determine if the proxified scope manager's `exit` method has been invoked or not. This is needed to disambiguate the case of premature scope exit `via` a return statement.

2. The result of user specified extant scope manager expressions are captured via the `WG_AUTOSIMULATOR_DETAIL_AUTOANY_EXPR_CAPTURE` macro. This is done so that such expressions can be specified without explicitly specifying their types and without having to use of Boost.Typeof to deduce their types. To implement this,

WG_AUTOSIMULATOR_DETAIL_AUTOANY_EXPR_CAPTURE borrows the techniques of Boost.Foreach.

Because this method of capture effectively type erases the captured object, extant_scopemngr_proxy_downcast must later be used to retrieve that object.

3. Adhoc scope managers are defined immediately before their use.