Declarative Control Flow

Prepared for CppCon
Bellevue, WA, Sep 20–25, 2015

Andrei Alexandrescu, Ph.D.
andrei@erdani.com

Introduction

There is no introduction slide

Challenge

- Define a transactional file copy function
- Either succeed
- . . . or fail successfully

Cheat Sheet

- From `boost::filesystem`:

```
void copy_file(
   const path& from,
   const path& to
);
```

Implementation

```
void copy_file_transact(const path& from,
      const path& to) {
   bf::path t = to.native() + ".deleteme";
   try {
      bf::copy_file(from, t);
      bf::rename(t, to);
   } catch (...) {
      ::remove(t.c_str());
      throw;
   }
}
```


Meh

- We now have explicit control flow
  - Not dedicated to core logic
- Of 8 lines, only 4 do “work”
- Y u play baseball?

Composition

- Define a transactional file *move* function
- Either succeed
- . . . or fail successfully

Implementation

```
void move_file_transact(const path& from,
      const path& to) {
   try {
      bf::copy_file_transact(from, to);
      bf::remove(from);
   } catch (...) {
      ::remove(to.c_str());
      throw;
   }
}
```

Ehm

- Same issues
- Also, functions that throw don't compose all that well

- Oops, there's a bug

Implementation (fixed)

```
void move_file_transact(const path& from,
      const path& to) {
   bf::copy_file_transact(from, to);
   try {
      bf::remove(from);
   } catch (...) {
      ::remove(to.c_str());
      throw;
   }
}
```

What Tools Do We Have?

- RAII tenuous
- `ScopeGuard` tenuous
- Composition only makes it worse
  - Series/nesting of `try`/`catch`
  - Urgh

- Meantime in CppCoreStandards:
  "E.18: Minimize the use of explicit `try`/`catch`"

Suddenly...

`goto fail;` doesn't look that bad anymore, eh?

Explicit Control Flow = Fail

Declarative Programming

- Focus on stating needed accomplishments
- As opposed to describing steps
- Control flow typically minimal/absent
- Execution is implicit, not explicit
- Examples: SQL, regex, make, config,. . .

- Let’s take a page from their book!

Surprising Insight

- Consider bona fide RAII with destructors:

✓ States needed accomplishment?
✓ Implicit execution?
✓ Control flow minimal?

- RAII *is* declarative programming!

More RAII: `ScopeGuard`

- Also declarative
- Less syntactic baggage than cdtors
- Flow is “automated” through placement
- Macro `SCOPE_EXIT` raises it to pseudo-statement status

Pseudo-Statement (old hat!)

```
namespace detail {
   enum class ScopeGuardOnExit {};
   template <typename Fun>
   ScopeGuard<Fun>
   operator+(ScopeGuardOnExit, Fun&& fn) {
      return ScopeGuard<Fun>(std::forward<Fun>(fn));
   }
}

#define SCOPE_EXIT \
   auto ANONYMOUS_VARIABLE(SCOPE_EXIT_STATE) \
   = ::detail::ScopeGuardOnExit() + [&]()
```


Preprocessor Trick (old hat!)

```
#define CONCATENATE_IMPL(s1, s2) s1##s2
#define CONCATENATE(s1, s2) CONCATENATE_IMPL(s1, s2)

#ifdef __COUNTER__
#define ANONYMOUS_VARIABLE(str) \
   CONCATENATE(str, __COUNTER__)
#else
#define ANONYMOUS_VARIABLE(str) \
   CONCATENATE(str, __LINE__)
#endif
```

Use (old hat!)

```
void fun() {
   char name[] = "/tmp/deleteme.XXXXXX";
   auto fd = mkstemp(name);
   SCOPE_EXIT { fclose(fd); unlink(name); };
   auto buf = malloc(1024 * 1024);
   SCOPE_EXIT { free(buf); };

   ...  use fd and buf ...
}
```

(if no ";" after lambda, error message is fail)

Painfully Close to Ideal!

```
⟨action1⟩
SCOPE_EXIT { ⟨cleanup1⟩ };
SCOPE_FAIL { ⟨rollback1⟩ }; // nope
⟨action2⟩
SCOPE_EXIT { ⟨cleanup2⟩ };
SCOPE_FAIL { ⟨rollback2⟩ }; // nope
```

One more for completeness

```
⟨action⟩
SCOPE_SUCCESS { ⟨celebrate⟩ };
⟨next⟩
```

- Powerful flow-declarative trifecta!
- Do not specify flow
- Instead declare circumstances and goals

~~May~~ ~~Will~~ Has become 100% portable: C++17 has it!

http://isocpp.org/files/papers/N4152.pdf

N4152

- "When in doubt, replace `bool` with `int`"

```
namespace std {
   bool uncaught_exception(); // old and bad
   int uncaught_exceptions(); // new and rad
}
```


Using `std::uncaught_exceptions`

```
class UncaughtExceptionCounter {
   int getUncaughtExceptionCount() noexcept;
   int exceptionCount_;
public:
   UncaughtExceptionCounter()
    : exceptionCount_(std::uncaught_exceptions()) {
   }
   bool newUncaughtException() noexcept {
      return std::uncaught_exceptions()
         > exceptionCount_;
   }
};
```

Layering

```
template <typename FunctionType, bool executeOnException>
class ScopeGuardForNewException {
   FunctionType function_;
   UncaughtExceptionCounter ec_;
public:
   explicit ScopeGuardForNewException(const FunctionType& fn)
         : function_(fn) {
   }
   explicit ScopeGuardForNewException(FunctionType&& fn)
         : function_(std::move(fn)) {
   }
   ~ScopeGuardForNewException() noexcept(executeOnException) {
      if (executeOnException == ec_.isNewUncaughtException()) {
         function_();
      }
   }
};
```


Icing

```
enum class ScopeGuardOnFail {};

template <typename FunctionType>
ScopeGuardForNewException<
   typename std::decay<FunctionType>::type, true>
operator+(detail::ScopeGuardOnFail, FunctionType&& fn) {
   return
      ScopeGuardForNewException<
            typename std::decay<FunctionType>::type, true>(
         std::forward<FunctionType>(fn));
}
```

Cake Candles

```
#define SCOPE_FAIL \
   auto ANONYMOUS_VARIABLE(SCOPE_FAIL_STATE) \
   = ::detail::ScopeGuardOnFail() + [&]() noexcept
```


Back to Example

Transactional Copy

```
void copy_file_transact(const path& from,
      const path& to) {
   bf::path t = to.native() + ".deleteme";
   SCOPE_FAIL { ::remove(t.c_str()); };
   bf::copy_file(from, t);
   bf::rename(t, to);
}
```

- All code *does work*

Transactional Move

```
void move_file_transact(const path& from,
      const path& to) {
   bf::copy_file_transact(from, to);
   SCOPE_FAIL { ::remove(to.c_str()); };
   bf::remove(from);
}
```

- Simpler than the no-exceptions version!

Please Note

Only `SCOPE_SUCCESS` may throw

Postconditions

```
int string2int(const string& s) {
   int r;
   SCOPE_SUCCESS {
      assert(int2string(r) == s);
   };
   ...
   return r;
}
```

Changing of the Guard

```
void process(char *const buf, size_t len) {
   if (!len) return;
   const auto save = buf[len - 1];
   buf[len - 1] = 255;
   SCOPE_EXIT { buf[len - 1] = save; };
   for (auto p = buf;;) switch (auto c = *p++) {
      ...
   }
}
```


Scoped Changes

```
bool g_sweeping;
void sweep() {
   g_sweeping = true;
   SCOPE_EXIT { g_sweeping = false; };
   auto r = getRoot();
   assert(r);
   r->sweepAll();
}
```

No RAII Type? No Problem!

```
void fileTransact(int fd) {
   enforce(flock(fd, LOCK_EX) == 0);
   SCOPE_EXIT {
      enforce(flock(fd, LOCK_UN) == 0);
   };
   ...
}
```

- No need to add a type for occasional RAII idioms

Remarks

- All examples taken from production code
- Declarative focus
  - Declare contingency actions by context
- `SCOPE_*` more frequent than `try` in new code
- The latter remains in use for actual handling
- Flattened flow
- Order still matters

Only detail left:

`std::uncaught_exceptions()`

Implemented and maintained on ALL major compilers

Credits

- Evgeny Panasyuk: compiler-specific bits `github.com/panaseleus/stack_unwinding`
- Daniel Marinescu: `folly` implementation `github.com/facebook/folly`
- Herb: got it in C++17
- Michael: Implemented it in Visual Studio 2015
- Ville Voutilainen implemented it for gcc

Summary

Summary

There is no summary slide