Systematic Error Handling in C++

Prepared for The C++ and Beyond Seminar 2012

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

Agenda

- Introduction
- `Expected<T>`
- `ScopeGuard11`

Introduction

What is Error Handling?

Means and techniques dedicated to handling possible but unexpected and useless inputs

Part of *Reliability*

- Error handling
  - working core
  - correct program
  - incorrect inputs
- General reliability
  - non-working core
  - incorrect program
  - incorrect inputs

Error Handling Scenarios

- User input errors
- File I/O errors
- Networking errors
- Environmental issues
- Device malfunctions
- . . .

Reliability Scenarios

- Program bugs
- Unexpected program state
- Corrupted I/O
- Core memory failure
- ...

Hypothesis

- Bad error handling engenders errors

Hypothesis

- Bad error handling engenders errors
- Insufficient testing makes improbable error scenarios never covered

Hypothesis

- Bad error handling engenders errors
- Insufficient testing makes improbable error scenarios never covered
- Legit errors lead to corrupt program

Hypothesis

- Bad error handling engenders errors
- Insufficient testing makes improbable error scenarios never covered
- Legit errors lead to corrupt program

Example: Thunderbird

- Handling of networking errors almost rigorously bad
- Modal error dialogs
- Numerous race conditions
- Metastable states
- Broken core invariants

`Expected` Types

Motivation

- Exceptions are, um, good
- Slow on the exceptional path
- Hopelessly serial
  - Only one exception in flight
  - Requires immediate, exclusive attention
  - Dedicated control flow
- Associated only with root reasons, not goals
  - “I/O error” doesn’t describe “saving weight file”

Key Insight

- Common complaint: “Error codes are limited! Exceptions are arbitrarily rich!”

- Make exceptions *be* the error codes!

Key Idea

`Expected<T>` is either a `T` or the exception preventing its creation

Instead of this...

```
// Returns 0 on error and sets errno. Ahem.
int parseInt(const string&);
```

. . . or this. . .

```
// Throws invalid_input or overflow
int parseInt(const string&);
```


. . . have this!

```
// Returns an expected int
Expected<int> parseInt(const string&);
```

Related Work

- Haskell: `Maybe T`
- Scala: `Option[T]`
- C#: `Nullable<T>`
- Boost: `optional<T>`

- Either a value of type `T`, or no value at all
- Defines primitives to store, test, extract
- Not packed with an exception

Related Work

- C++11: `promise<T>`/`future<T>`

- Either a value of type `T`, or an exception
- Primitives focused on inter-thread, async communication
- We want eager, synchronous

`Expected<T>` characteristics

- Associates errors with computational goals
- Naturally allows multiple exceptions in flight
- Switch between "error handling" and "exception throwing" styles
- Teleportation possible
  - Across thread boundaries
  - Across `nothrow` subsystem boundaries
  - Across time: save now, throw later
- Collect, group, combine exceptions

`Expected<T>` implementation

- Basic idea: `Variant` with a `T` or a `std::exception_ptr`

```
template <class T> class Expected {
   union {
      T ham;
      std::exception_ptr spam;
   };
   bool gotHam;
   Expected() {} // used internally
public:
```

Expected<T> implementation

```
Expected(const T& rhs) : ham(rhs), gotHam(true) {}
Expected(T&& rhs)
   : ham(std::move(rhs))
   , gotHam(true) {}
Expected(const Expected& rhs) : gotHam(rhs.gotHam) {
   if (gotHam) new(&ham) T(rhs.ham);
   else new(&spam) std::exception_ptr(rhs.spam);
}
Expected(Expected&& rhs) : gotHam(rhs.gotHam) {
   if (gotHam) new(&ham) T(std::move(rhs.ham));
   else new(&spam)
      std::exception_ptr(std::move(rhs.spam));
}
```

```
void swap(Expected& rhs) {
    if (gotHam) {
        if (rhs.gotHam) {
            using std::swap;
            swap(ham, rhs.ham);
        } else {
            auto t = std::move(rhs.spam);
            new(&rhs.ham) T(std::move(ham));
            new(&spam) std::exception_ptr(t);
            std::swap(gotHam, rhs.gotHam);
        }
    } else {
        if (rhs.gotHam) {
            rhs.swap(*this);
        } else {
            spam.swap(rhs.spam);
            std::swap(gotHam, rhs.gotHam);
        }
    }
}
```

Building from exception

```
template <class E>
static Expected<T> fromException(const E& exception) {
   if (typeid(exception) != typeid(E)) {
      throw std::invalid_argument(
         "slicing detected");
   }
   return fromException(
      std::make_exception_ptr(exception));
}
```

Building from exception

```
static Expected<T> fromException(std::exception_ptr p) {
   Expected<T> result;
   result.gotHam = false;
   new(&result.spam) std::exception_ptr(std::move(p));
   return result;
}

static Expected<T> fromException() {
   return fromException(std::current_exception());
}
```

Access

```
bool valid() const {
   return gotHam;
}

T& get() {
   if (!gotHam) std::rethrow_exception(spam);
   return ham;
}
const T& get() const {
   if (!gotHam) std::rethrow_exception(spam);
   return ham;
}
```

Probing for Type

```
template <class E>
bool hasException() const {
   try {
      if (!gotHam) std::rethrow_exception(spam);
   } catch (const E& object) {
      return true;
   } catch (...) {
   }
   return false;
}
```

Icing

```
template <class F>
static Expected fromCode(F fun) {
   try {
      return Expected(fun());
   } catch (...) {
      return fromException();
   }
}

auto r = Expected<string>::fromCode([] {
   ...
});
```

Use example (callee)

```
Expected<int> parseInt(const std::string& s) {
   int result;
   ...
   if (nonDigit) {
      return Expected<int>::fromException(
         std::invalid_argument("not a number"));
   }
   ...
   if (tooManyDigits) {
      return Expected<int>::fromException(
         std::out_of_range("overflow"));
   }
   ...
   return result;
}
```

Use example (caller)

```
// Caller
string s = readline();
auto x = parseInt(s).get(); // throw on error
auto y = parseInt(s); // won't throw
if (!y.valid()) {
   // handle locally
   if (y.hasException<std::invalid_argument>()) {
      // no digits
      ...
   }
   y.get(); // just "re"throw
}
```


`Expected<T>` aftermath

- Encodes a value OR the error in attempting to produce it
- Groups data with error flow
- Non-serial in nature
- Supports different styles of coding
- Arbitrarily rich “error codes”

ScopeGuard11

Recall `ScopeGuard`

- Introduced in 2000 article
- Makes transactional code linear
- Lightweight RAII
- Scales well to multi-step transactions

⟨action⟩

$\langle$`action`$\rangle$

$\langle$`cleanup`$\rangle$

⟨action⟩

⟨cleanup⟩

⟨next⟩

$\langle$`action`$\rangle$

$\langle$`cleanup`$\rangle$
$\langle$`next`$\rangle$
$\langle$`rollback`$\rangle$

C

```
if (⟨action⟩) {
    if (!⟨next⟩) {
        ⟨rollback⟩
    }
    ⟨cleanup⟩
}
```

C+

```
class RAII {
    RAII() { ⟨action⟩ }
    ~RAII() { ⟨cleanup⟩ }
};
...
RAII raii;
try {
    ⟨next⟩
} catch (...) {
    ⟨rollback⟩
    throw;
}
```

Entering Composition

C (expand ⟨next$_1$⟩)

```
if (⟨action₁⟩) {
    if (⟨action₂⟩) {
        if (!⟨next₂⟩) {
            ⟨rollback₂⟩
            ⟨rollback₁⟩
        }
        ⟨cleanup₂⟩
    } else {
        ⟨rollback₁⟩
    }
    ⟨cleanup₁⟩
}
```

C+

```
class RAII1 {
    RAII1() { ⟨action₁⟩ }
    ~RAII1() { ⟨cleanup₁⟩ }
};
class RAII2 {
    RAII2() { ⟨action₂⟩ }
    ~RAII2() { ⟨cleanup₂⟩ }
};
...
```

C+ (expand $\langle$next$_1\rangle$)

```
RAII1 raii1;
try {
    RAII2 raii2;
    try {
        ⟨next₂⟩
    } catch (...) {
        ⟨rollback₂⟩
        throw;
    }
} catch (...) {
    ⟨rollback₁⟩
    throw;
}
```

Dislocation + Nesting = Fail

C++11 with `ScopeGuard`

```
⟨action⟩
auto g1 = scopeGuard([] { ⟨cleanup⟩ });
auto g2 = scopeGuard([] { ⟨rollback⟩ });
⟨next⟩
g2.dismiss();
```

ScopeGuard composition

```
⟨action₁⟩
auto g1 = scopeGuard([&] { ⟨cleanup₁⟩ });
auto g2 = scopeGuard([&] { ⟨rollback₁⟩ });
⟨action₂⟩
auto g3 = scopeGuard([&] { ⟨cleanup₂⟩ });
auto g4 = scopeGuard([&] { ⟨rollback₂⟩ });
⟨next₂⟩
g2.dismiss();
g4.dismiss();
```

Macro Edition

```
⟨action₁⟩
SCOPE_EXIT { ⟨cleanup₁⟩ };
auto g1 = scopeGuard([&] { ⟨rollback₁⟩ });
⟨action₂⟩
SCOPE_EXIT { ⟨cleanup₂⟩ };
auto g2 = scopeGuard([&] { ⟨rollback₂⟩ });
⟨next₂⟩
g2.dismiss();
g1.dismiss();
```

Painfully Close to Ideal!

```
⟨action₁⟩
SCOPE_EXIT { ⟨cleanup₁⟩ };
SCOPE_FAIL { ⟨rollback₁⟩ }; // nope
⟨action₂⟩
SCOPE_EXIT { ⟨cleanup₂⟩ };
SCOPE_FAIL { ⟨rollback₂⟩ }; // nope
⟨next₂⟩
```

How does `ScopeGuard` work?

- C++98: elaborate implementation featuring type erasure
- In C++11 we can exploit
  - Type inference and `auto`
    - No need for type erasure
  - Lambda functions
    - Defer arbitrary code
  - Move semantics
    - No spurious double cleanup

Implementation

```
template <class Fun>
class ScopeGuard {
    Fun f_;
    bool active_;
public:
    ScopeGuard(Fun f)
          : f_(std::move(f))
          , active_(true) {
    }
    ~ScopeGuard() { if (active_) f_(); }
    void dismiss() { active_ = false; }
```

. . . to be continued . . .

Implementation (cont'd)

. . . continued . . .

```
    ScopeGuard() = delete;
    ScopeGuard(const ScopeGuard&) = delete;
    ScopeGuard& operator=(const ScopeGuard&) = delete;
    ScopeGuard(ScopeGuard&& rhs)
            : f_(std::move(rhs.f_))
            , active_(rhs.active_) {
       rhs.dismiss();
    }
};
```

Type Deduction

```
template <class Fun>
ScopeGuard<Fun> scopeGuard(Fun f) {
   return ScopeGuard<Fun>(std::move(f));
}
```

Use

```
void fun() {
    char name[] = "/tmp/deleteme.XXXXXX";
    auto fd = mkstemp(name);
    auto g1 = scopeGuard([] {
        fclose(fd);
        unlink(name);
    });
    auto buf = malloc(1024 * 1024);
    auto g2 = scopeGuard([] { free(buf); });

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

Pseudo-Statement

```
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 Wonders

```
#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
```

Lo and Behold

```
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 meh)

Summary

Summary

- Make error handling simple and systematic
- `ExpectedT` encapsulates values with their history of failure
- `ScopeGuard11` encapsulates scoped control flow