Expect the `expected`

University of Chicago

Andrei Alexandrescu, Ph.D.

andrei@erdani.com

2020-04-25

Exceptions, yay!. . . ?

- Most of us took them non-critically
- “Here’s the construct. . . use it”
- What’s a proper baseline?
- What were their design goals?
- What were their intended use cases?
- How do their semantics support the use cases?
- What were the consequences of their design?
- How to write code playing on their strengths?

So many to Choose From

- “One man’s constant is another man’s variable”—Alan Perlis
- “One person’s fatal error is another person’s common case”—Anonymized

We Want Errors!

- General: learn once use many
- Minimize soft errors; maximize hard errors
  - Avoid metastable states
- Allow centralized handling
  - Keep error handling out of most code
- Allow local handling
  - Library can’t decide handling locus
- Transport an arbitrary amount of error info
- Demand little cost on the normal path
- Make correct code easy to write

Inventing Exceptions

```
int atoi(const char * s);
```

- What’s wrong with it?
  - Returns zero on error
  - “0”, “ 0”, “ +000 ” are all valid inputs
  - Zero is a commonly-encountered value
  - `atoi` is a surjection

- Distinguish valid from invalid input a posteriori is almost as hard as a priori!

`errno`

- + General
- - Minimize soft errors
- + Centralized handling
- + Local handling
- - Arbitrary amount of error info
- + Little cost on the normal path
- - Make correct code easy to write
  - Error handling entirely optional
  - Threading issues

Special Value

- - General (won’t work with surjective functions)
- - Minimize soft errors
- - Centralized handling
- + Local handling
- - Arbitrary amount of error info
- ? Little cost on the normal path
- - Make correct code easy to write
    - Error handling often optional
    - Error handling code intertwined with normal code

Value of Separate Type

- + General
- ? Minimize soft errors
- - Centralized handling
- + Local handling
- + Arbitrary amount of error info
- + Little cost on the normal path
- - Make correct code easy to write
    - Error handling requires extra code & data

```
long strtol(const char*s, const char**e,
    int r);
```

Exceptions?

- We want to pass arbitrary error info around:

```
class invalid_input { ... };
int|invalid_input atoi(const char * str);
int|invalid_input r = atoi(some_string);
typeswitch (r) {
   case int x { ... }
   case invalid_input err { ... }
};
```

- Hat tip: algebraic types

Exceptions? (cont'd)

- We want to allow centralized error handling
  - Break the typeswitch ⇒ covert return types!

```
expected<int>|unexpected<invalid_input>
   atoi(const char*);
```

- Local code should afford to ignore `invalid_input`
  - ⇒ A function has an overt return type plus one or more covert return types
- Q: Where do the covert return values go?

Exceptions? (cont'd)

- Covert values must "return" to a caller upper in the dynamic invocation chain
- Only certain callers understand certain errors
- $\Rightarrow$ Covert returned types come together with covert execution paths!
- $\Rightarrow$ Callers plant return points collecting such types
- $\Rightarrow$ Type-based, first-match exception handling

Exceptions: Aftermath

- + General
- ? Minimize soft errors
- + Centralized handling
- − Local handling
- + Arbitrary amount of error info
- + Little cost on the normal path
- ? Make correct code easy to write
  - 2000: yes
  - 2008: no
  - 2020: maybe

“[. . .] error handling idioms and practices remain contentious and confusing within the C++ community (as well as within the Boost community).”

— Charley Bay

Top Issues with Exceptions

- Metastable states
  - User must ensure transactional semantics
    - Destructors
    - `ScopeGuard`
- Local error handling unduly hard/asymmetric
- Hard to analyze
  - By human and by machine
  - Too many paths!
- Odd semantics
  - Composition is tenuous
  - Can’t have more than one exception
  - Except when we can

Today's Plan

- + Local handling
- + Minimize soft errors
- + Make correct code easier to write

- Must start with a few background items

Background Technologies

- `std::variant` (C++17) or `boost::variant`
  - Gives equal importance to all members
- `std::optional` (C++17), `boost::optional`
  - No extra information in the "null" state
- More exotic: the Maybe/Either monads

- Painfully close to what's needed!

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

Union Types

- Discriminated unions
- Defined by e.g. `boost::any`, `variant`
- Typical implementation:

```
template <class T, class U> class Either {
    union { // Changed in C++11
        T t_;
        U u_;
    } data_;
    bool isT;
    ...
};
```

`std::expected<T, E>`

- Idea: We want to express the union of an overt type and a covert type
- Normal case: value of overt type is there
- Erroneous case: an `E` is there
  - It has extra info about what happened!
  - Is eagerly constructed, lazily thrown!

An `expected<T, E>` is either a `T` or an explanation `E` on why the `T` couldn't be produced.

See proposal `https://wg21.link/P0323` by Vicente Botet, JF Bastien

Flexibility

- Unify local and centralized error handling: `expected<int, err> atoi(const char *);`
- Wanna local? Check `result.has_value()`, `result.error()`
  - Idiom: `if (result) use(*result);`
- Wanna centralized? Just use `*result`
  - That is an `int`, or throws `err` if not

`expected<T, E>` 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

Implementation (partial)

```
template<class T, class E> class expected {
    union { T yay; E nay; };
    bool ok = true;
public:
    expected() { new(&yay) T(); }
    expected(const T& rhs) { new(&yay) T(rhs); }
    expected(const unexpected<E>& rhs) : ok(false) {
        new(&nay) E(rhs.value());
    }
    template<class U = T> explicit expected(U&& rhs) {
        new(&yay) T(forward<U>(rhs));
    }
    ...
};
```

```cpp
expected(const expected& rhs) : ok(rhs.ok) {
   if (ok) new(&yay) T(rhs.yay);
   else new(&nay) E(rhs.nay);
}
expected(expected&& rhs) : ok(rhs.ok) {
   if (ok) new(&yay) T(std::move(rhs.yay));
   else new(&nay) E(std::move(rhs.nay));
}
```

```
T& operator*() {
   //if (!ok) throw nay;
   return yay; // UB galore
}
const T& operator*() const;
T&& operator*() &&;
const T&& operator*() const &&;

T* operator->() { return &**this; }
const T* operator->() const;

const E& error() const {
   assert(!ok);
   return nay;
}
E& error();
E&& error() &&;
const E&& error() const &&;
```

```cpp
bool has_value() const noexcept {
    return ok;
}
explicit operator bool() const noexcept {
    return ok;
}

T& expected::value() & {
    if (!ok) throw nay;
    return yay;
}
const T& expected::value() const&;
T&& expected::value() &&;
const T&& expected::value() const&&; // any useful scenario?

// Returns value() if ok, T(forward<U>(v)) otherwise
template <class U>
T value_or(U&& v) const&;
template <class U>
T value_or(U&& v) &&;
```

```
enable_if_t<is_nothrow_move_constructible_v<T>
        && is_swappable_v<T&>
        && is_nothrow_move_constructible_v<E>
        && is_swappable_v<E&>>
swap(expected& rhs) {
    if (ok) {
        if (rhs.ok) {
            using std::swap;
            swap(yay, rhs.yay);
        } else {
            rhs.swap(*this);
        }
    } else {
        if (!rhs.ok) {
            using std::swap;
            swap(nay, rhs.nay);
        } else {
            ...
        }
    }
}
```

The odd part of `swap`

```
// ... ok=false, rhs.ok=true ...
E t{std::move(nay)};
nay.~E();
new(&yay) T(std::move(rhs.yay));
ok = true;
rhs.yay.~T();
new(&rhs.nay) E(std::move(t));
rhs.ok = false;
```

Typical use

```
expected<double, runtime_error> good = 100.0;
assert(*good == 100);
// unexpected disambiguates "bad" case
expected<double, runtime_error> bad =
    unexpected(runtime_error("!"));
assert(!bad.has_value());
```

Typical use (function definition)

```
expected<double, runtime_error> relative(double a, double b) {
    if (a == 0)
        return unexpected(
            runtime_error("Cannot compute relative to 0"));
    return (b - a) / a;
}
```

Using `expected<T, E>`: Centralized

- Centralized error handling: convert `expected<T, E>` to `T&` by using `operator*`
- `E` is thrown if the object is a dud
- Code is similar to that with entirely covert returns

  ```
  double growth = *relative(1.00, x);
  ```
- Separate normal path from error path
- Just like with exceptions—just add a `*`!

Using `expected<T, E>`: Local

- Localized error handling:

```
expected<int, err> r = atoi(some_string);
if (!r) {
    ... local error handling ...
}
```

- Just like good ol' error handling with special values
- Exacts a tad more cost
- No more issues with surjections $\Rightarrow$ general!

Tree n da Forest

- If:
  - A `expected<T, E>` object is a dud &&
  - Nobody attempts to dereference it...
- Then:
  - No harm done!

Tree n da Forest

- If:
  - A `expected<T, E>` object is a dud &&
  - Nobody attempts to dereference it. . .
- Then:
  - No harm done!

- (Unless you're LLVM)

Checkpoint

- Associates errors with computational goals.
- Naturally allows multiple errors in flight.
- Teleportation possible.
- Across thread boundaries.
- Across no-throw subsystem boundaries.
- Across time: save now, throw later.
- Collect, group, combine errors.
- Much simpler for a compiler to optimize.

Thank You!

```
speaker.~Speaker();
```