Choose Your Poison: Exceptions or Error Codes?

Andrei Alexandrescu
Petru Marginean

1

Agenda

- Exceptions: WTF?
  - Why The Frenzy?
- Top 3 problems with exceptions
- Help from the type system
  - The None type, type erasure, and variants, oh my!
- The `Likely<T>` type
- Conclusions

2

Exceptions: teleology

- 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?

A Case for Dual Errors

- "One man's constant is another man's variable"
  - Alan Perlis
- "One person's fatal error is another person's common case"
  - W.P.P.

Desiderata

- 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!

Inventing Exceptions

- Four solutions for returning error information:
    1. Set global state
        - The errno approach
    2. Encode error information as a special returned value
        - the out-of-band value approach
    3. Encode error information as a value of a distinct type
        - the “error code return” approach
    4. Exceptions

7

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

8

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 much extra code & data
  - 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 { … }
};
```


Exceptions? (cont'd)

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

```
overt<int>|covert<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
- => Covert returned types come together with covert execution paths!
- => Callers plant return points collecting such types
- => Type-based, first-match exception handling

13

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
  - 1987: yes
  - 1997: no
  - 2007: maybe

14

Top 3 Issues with Exceptions

- Metastable states
  - User must ensure transactional semantics
    - Destructors
    - `ScopeGuard`
- Local error handling unduly hard/asymmetric
  - By-value semantics prevent library approaches
  - Can't say `GuardedCall(Function(args))`
- Hard to analyze
  - By human and by machine

Today

- + Local handling
- + Minimize soft errors
- + Make correct code easier to write
- Must start with a few background items

1. The None type

- Returned by a function with no overt returns:

```
None Abort();
None Exit(int code);
None LoopForever();
```

- Properties:
  - Can be substituted for any type
    - The bottom of the type hierarchy
  - Destructor throws
  - Noncopyable

17

2. Type Erasure

- Cloaks an arbitrarily typed object under a uniform interface
- Used in e.g. ScopeGuard, boost::dynamic_any
- Typical implementation:

```
class Cloak {
  auto_ptr<Interface> p_;
public:
  template <class T> Cloak(const T& t)
    : p_(new InterfaceImpl<T>(t)) {}
  ...
};
```

18

3. Union Types

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

```
template <class T, class U> class Variant {
  union {
    char[appropriate_size] buf_;
    AppropriateAlignType align_;
  } data_;
  bool isT;
  ...
};
```


Likely<T>

- 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: a value akin to None is there
  - None has extra info using *type erasure*!
- Unify local and central error handling

```
Likely<int> atoi(const char *);
```

- Wanna local? Check Likely<T>::HasValue()
- Wanna centralized? Use Likely<T> as you'd use a T

Creating Likely<T>

```
template <typename T> struct Likely {
  Likely();
  Likely(const T& v);
  Likely(const Likely& obj);
  Likely& operator=(const Likely&);
  ~Likely() throw(something);
  enum InvalidT { Invalid };
  template <typename E>
  Likely(const E& obj, InvalidT);
  operator T&();
  operator const T&() const;
  bool HasValue() const;
  template <typename E> const E* Probe() const;
private:
  Variant<T, auto_ptr<CovertInterface> > data_;
};
```

21

Using Likely<T>: Centralized

- Centralized error handling: convert Likely<T> to T& liberally
- Exception is thrown if the object is a dud
- Code is similar to that with entirely covert returns

```
int x = atoi(some_string);
```

- Separate normal path from error path
- Just like with exceptions

22

Using Likely<T>: Local

- Localized error handling:

```
Likely<int> r = atoi(some_string);
if (r.HasValue()) {
  auto p = r.Probe<ConvException>();
  ... local error handling ...
}
```

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

Using Likely<T>: Ignoramus

- If:
  - A `Likely<T>` object is a dud &&
  - Nobody attempts to dereference it &&
  - Nobody checks `IsValue()` &&
  - `!std::uncaught_exception()`
- Then:
  - `Likely<T>`'s destructor throws an exception (ouch!)
- Keeps error handling required
- Avoids metastable states
- Easy to supress: `IGNORE_ERROR(atoi(str));`

The Covert Side

```
struct CovertInterface {
    virtual ~CovertInterface() throw() {}
    virtual void Throw() const = 0;
    virtual bool IsEnabled() const = 0;
    virtual void Disable() = 0;
};
```

25

The Covert Side

```
template <typename E> struct Covert : CovertInterface, E
    {
  Covert(const E& obj) : E(obj), enabled(true) {}
  virtual void Throw() const {
    if (!std::uncaught_exception())
      throw static_cast<const E&>(*this);
  }
  virtual bool IsEnabled() const { return enabled; }
  virtual void Disable() { enabled = false; }
private:
  Covert(const Covert&);
  Covert& operator =(const Covert&);
  bool enabled;
};
```

26

The MI trick

- Multiple inheritance allows implementing Probe

```
template <typename E>
const E* Likely<T>::Probe() const {
  auto p = dynamic_cast<E*>(getCovertPtr());
  if (!p) return 0;
  const E& e = *pPtr;
  return &e;
};
```

27

Conclusions

- Exceptions' design address a complicated web of desiderata
  - Fails to provide complete solution
  - Better than others
  - Requires a shift in code writing style
- Possible to make local and central error handling interchangeable
  - Type system can help
  - Keeps error handling required
  - Avoid asymmetry

28