PlayStation®

Optional, Expected, Error, Oh My!

LLVM Developers’ Meeting, October 2023

Paul T Robinson

Sony
Interactive
Entertainment

Our Example Specification

Given integers X and Y, if X = N x Y for some integer N, return N

- Like many specifications, it is incomplete. If X != N x Y, what happens?
- But it’s the specification we have.
- In developing a function that satisfies this specification, we must make choices about error handling.

Error Handling Option 0: Ignore The Problem

```
// If x is an integer multiple of y, return that multiplier.
int getExactQuotient(int x, int y) {
  return x / y;
}
```

If X isn’t an integer multiple of Y, you get some (unspecified) number.

This is probably not what the caller really wants. Pesky callers.

If Y == 0, this is UB, and probably will crash the program.

Can you believe some humans don’t want their programs to crash?? Pesky humans.

Error Handling Option 0.5: Use Normal C++ Error Handling: throw an exception

```
// If x is an integer multiple of y, return that multiplier.
int getExactQuotient(int x, int y) {
  if (y == 0) throw “Ambiguous value of Y”;
  if (x % y != 0) throw “Not an integer multiple”;
  return x / y;
}
```

If X isn’t an integer multiple of Y, or Y == 0, throw an exception.

This is what everyone learns in C++ class.

LLVM coding standard forbids exceptions. Pesky coding standards.

Error Handling Option 1: Use An In-Band Value

```
// If x is an integer multiple of y, return that multiplier.
// Otherwise, return 0.
int getExactQuotient(int x, int y) {
  if (y == 0) return 0;
  if (x % y != 0) return 0;
  return x / y;
}
```

Return value of 0 now means THREE different things, one of which is a valid result.

There is no in-band value that cannot be a valid result (in this specification). Might be okay.

Most common idiom when returning a pointer to something, `nullptr` => no such object.

Error Handling Option 1: Use An In-Band Value [caller]

```
assert(getExactQuotient(0, 0) == 0); // Incorrect param => 0.
assert(getExactQuotient(1, 2) == 0); // Inexact => 0.
assert(getExactQuotient(0, 2) == 0); // Exact multiple => 0.
```

Error Handling Option 2: Use A `bool` Return Type

```
// If x is an integer multiple of y, pass back that multiplier.
// The bool return value indicates whether it’s valid.
bool getExactQuotient(int x, int y, int &m) {
  if (y == 0) return false;
  if (x % y != 0) return false;
  m = x / y;
  return true;
}
```

Return value says valid or not. “Not” doesn’t indicate why not. Might be okay.

Caller needs to manage return value and additional parameter for result.

Error Handling Option 2: Use A `bool` Return Type [caller]

```
int M = -1;
assert(!getExactQuotient(0, 0, M));          // Incorrect param => false.
assert(!getExactQuotient(1, 2, M));          // Inexact => false.
assert(getExactQuotient(0, 2, M) && M == 0); // Exact multiple => 0.
```

Error Handling Option 3: Use `std::optional<T>`

Like a `std::pair<T, bool>`

```
#include <optional>
// If x is an integer multiple of y, return that multiplier.
std::optional<int> getExactQuotient(int x, int y) {
  if (y == 0) return std::nullopt;     // Ambiguous Y.
  if (x % y != 0) return std::nullopt; // Not exact multiple.
  return x / y;
}
```

Return value of 0 is unambiguously a valid result!

False result doesn’t distinguish why. Might be okay.

Very common idiom for non-pointer return types.

Error Handling Option 3: Use `std::optional<T>` [caller]

```
assert(!getExactQuotient(0, 0));         // Incorrect param => false.
assert(!getExactQuotient(0, 0).has_value()); // ditto
auto I = getExactQuotient(1, 2);         // Inexact => false; caller can
assert(I.value_or(-1) == -1);            //   provide a default.
assert(*getExactQuotient(0, 2) == 0);    // Exact multiple => true, value 0;
                                         //  but unchecked status, possible UB.
auto Q = getExactQuotient(0, 2);
assert(Q.has_value() && Q.value() == 0); // Check status before fetching result.
```

Error Handling Option 4: Use `llvm::ErrorOr<T>`

Like a `std::pair<T, std::error_code>`

```
#include "llvm/Support/ErrorOr.h"
// If x is an integer multiple of y, return that multiplier.
llvm::ErrorOr<int> getExactQuotient(int x, int y) {
  if (y == 0) return std::errc::argument_out_of_domain;
  if (x % y != 0) return std::errc::result_out_of_range;
  return x / y;
}
```

False result identifies why (if you pick different `std::error_code` values).

Must stick with the standard error codes.

NAMING FAIL: This is NOT `std::pair<T, llvm::Error>` (that’s called `llvm::Expected<T>`)

Error Handling Option 4: Use llvm::ErrorOr<T> [caller]

```
assert(!getExactQuotient(0, 0));      // Incorrect param => false.
// if (*getExactQuotient(0, 0) == 0)  // Fetching nonexistent value
//   return 1;                        //  guaranteed assert, unlike optional<T>
auto Result = getExactQuotient(1, 2); // Inexact. More typical checking code is:
if (std::error_code ec = Result.getError())
  return 1;                           // (real code probably would “return ec;”).
assert(*getExactQuotient(0, 2) == 0); // Exact multiple; can fetch value if it
                                      //  exists (but asserts if you’re wrong).
assert(getExactQuotient(0, 2).get() == 0); // ditto
```

Error Handling Option 5: Use `llvm::Error`

Returns a status code. Actual result is separate. Caller MUST check the Error.

```
#include "llvm/Support/Error.h"
// If x is an integer multiple of y, pass back that multiplier.
llvm::Error getExactQuotient(int x, int y, int &m) {
  if (y == 0)
    return llvm::make_error<llvm::StringError>(
        llvm::inconvertibleErrorCode(), “Y is zero”);
  if (x % y != 0)
    return llvm::make_error<llvm::StringError>(
        llvm::inconvertibleErrorCode(), “Quotient is not integer”);
  m = x / y;
  return llvm::Error::success();
}
```

Error Handling Option 5: Use `llvm::Error [caller]`

```
int M = -1;
if (auto Result = getExactQuotient(0, 0, M))
  // True => error state, but still doesn’t count as checked.
  llvm::handleAllErrors(std::move(Result), // Do something intelligent here.
    [](const llvm::StringError &Err) { printf("%s\n", Err.getMessage().c_str()); });
llvm::consumeError(ExactQuotient(1, 2, M)); // Rarely okay to ignore Error.
if (auto Result = getExactQuotient(0, 2, M))
  llvm::report_fatal_error(std::move(Result));
else
  // False => success, and sufficiently checked.
  assert(M == 0);
llvm::cantFail(getExactQuotient(0, 2, M)); // llvm_unreachable if not success.
```

Error Handling Option 6: Use `llvm::Expected<T>`

Like a `std::pair<T, llvm::Error>`. Caller MUST check the Error.

```
#include "llvm/Support/Error.h" // Not Expected.h!
// If x is an integer multiple of y, return that multiplier.
llvm::Expected<int> getExactQuotient(int x, int y) {
  if (y == 0)
    return llvm::make_error<llvm::StringError>(
        llvm::inconvertibleErrorCode(), “Y is zero”);
  if (x % y != 0)
    return llvm::make_error<llvm::StringError>(
        llvm::inconvertibleErrorCode(), “Quotient is not integer”);
  return x / y;
}
```

Error Handling Option 6: Use `llvm::Expected<T> [caller]`

```
auto Result = getExactQuotient(0, 0);
if (auto E = Result.takeError())
  // True => error state, but still doesn’t count as checked.
  llvm::handleAllErrors(std::move(E), // Do something intelligent here.
    [](const llvm::StringError &Err) { printf("%s\n", Err.getMessage().c_str()); });
if (Result = getExactQuotient(0, 2))
  // Expected<T> true => success, similar to std::optional.
  assert(*Result == 0);
else
  llvm::report_fatal_error(std::move(Result.takeError()));
```

Summary

For an API returning type T, typical error handling choices

Return type	**Includes value?**	**Error detail?**	**Mandatory check?**
T (in-band value)	Yes	Maybe (for enum)	No
bool	No	No	No
std::optional<T>	Yes	No	No
llvm::ErrorOr<T>	Yes	Yes (std::errc)	No
llvm::Error	No	Yes (custom)	Yes
llvm::Expected<T>	Yes	Yes (custom)	Yes