Expectify

Rich Polymorphic Error Handling with llvm::Expected<T>

Stefan Gränitz
Freelance Dev C++ / Compilers / Audio

C++ User Group Berlin
19. September 2017

No answer for that..

YES! Exceptions

NO Exceptions

Error Handling in the exception-free codebase

Whats the matter?

- ad-hoc approaches to indicate errors
- return `bool`, `int`, `nullptr` or `std::error_code`
- no concept for context information
- made for enumerable errors
- suffer from lack of enforcement

C++ has an answer for this

- a[illegible]es to indicate err[illegible]
- re[illegible], `int`, `nullptr` or `std::er`[illegible]ode
- no concept for context information
- made [illegible]le errors
- suffe[illegible]orcemen[illegible]

Exceptions

type-safe handlers

user-defined error types

total enforcement

How to get these benefits without using exceptions?

```
Error foo(...);
Expected<T> bar(...);
```

Polymorphic Error as a Return Value scheme

Idiomatic usage

```
Error foo(...);

// conversion to bool "checks" error
if (auto err = foo(...))
  return err; // error case

// success case
```

Idiomatic usage

```
Error foo(...);
Expected<T> bar(...);

foo(...); // unchecked Error triggers abort
bar(...); // so does unchecked Expected
```

Idiomatic usage

```
Error foo(...);
Expected<T> bar(...);

// errors can only be moved, not copied
Error err1 = foo(...);
Error err2 = std::move(err1);

// ... same for Expected ...
```

Don't silently
disappear or duplicate
(like Exceptions)

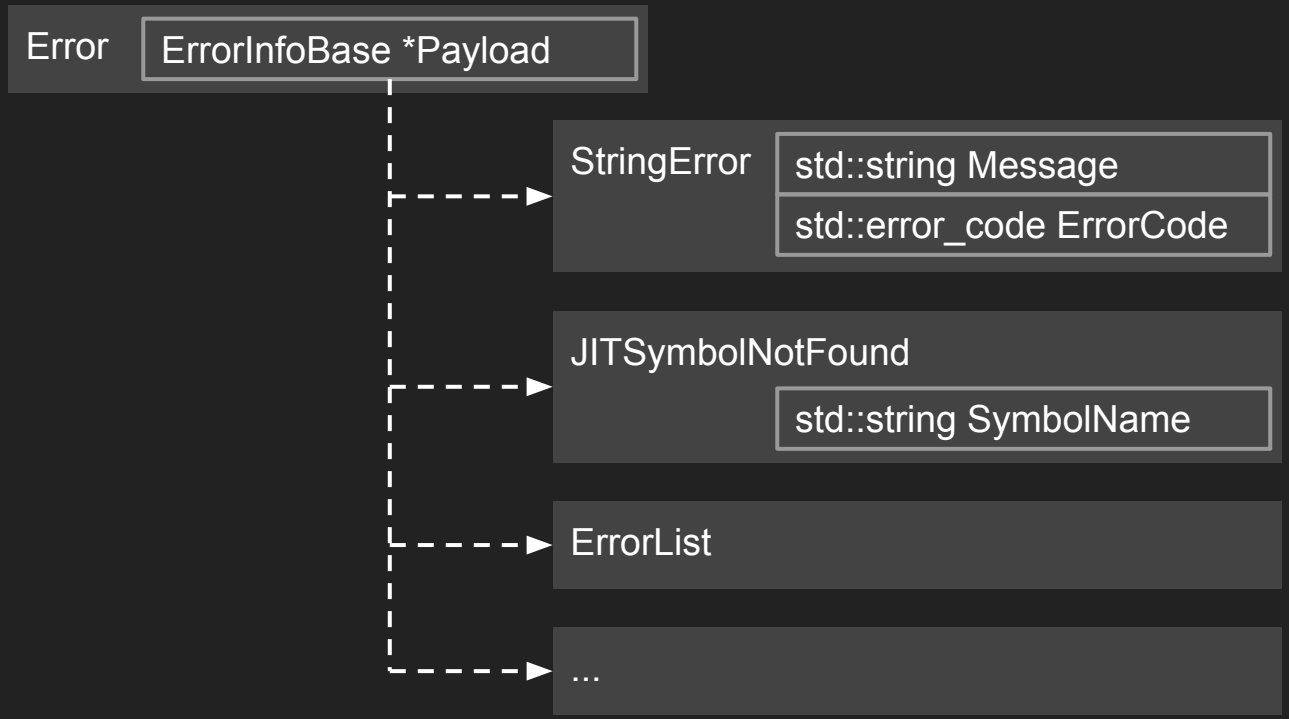
Interface

```
class ErrorInfoBase {
public:
  virtual ~ErrorInfoBase() = default;

  /// Print an error message to an output stream.
  virtual void log(std::ostream &OS) const = 0;

  /// Return the error message as a string.
  virtual std::string message() const;

  /// Convert this error to a std::error_code.
  virtual std::error_code convertToErrorCode() const = 0;
};
```

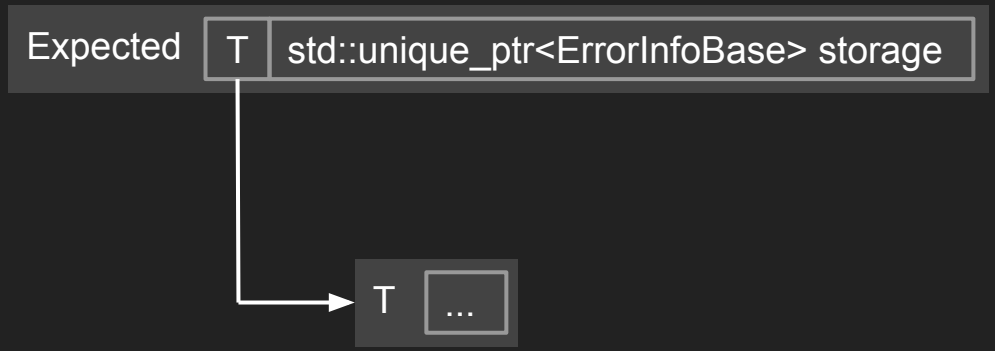
Implementation

```
class StringError : public ErrorInfo<StringError> {
public:
  static char ID;

  StringError(std::string Msg, std::error_code EC);

  void log(std::ostream &OS) const override;
  std::error_code convertToErrorCode() const override;

  const std::string &getMessage() const { return Msg; }

private:
  std::string Msg;
  std::error_code EC;
};
```

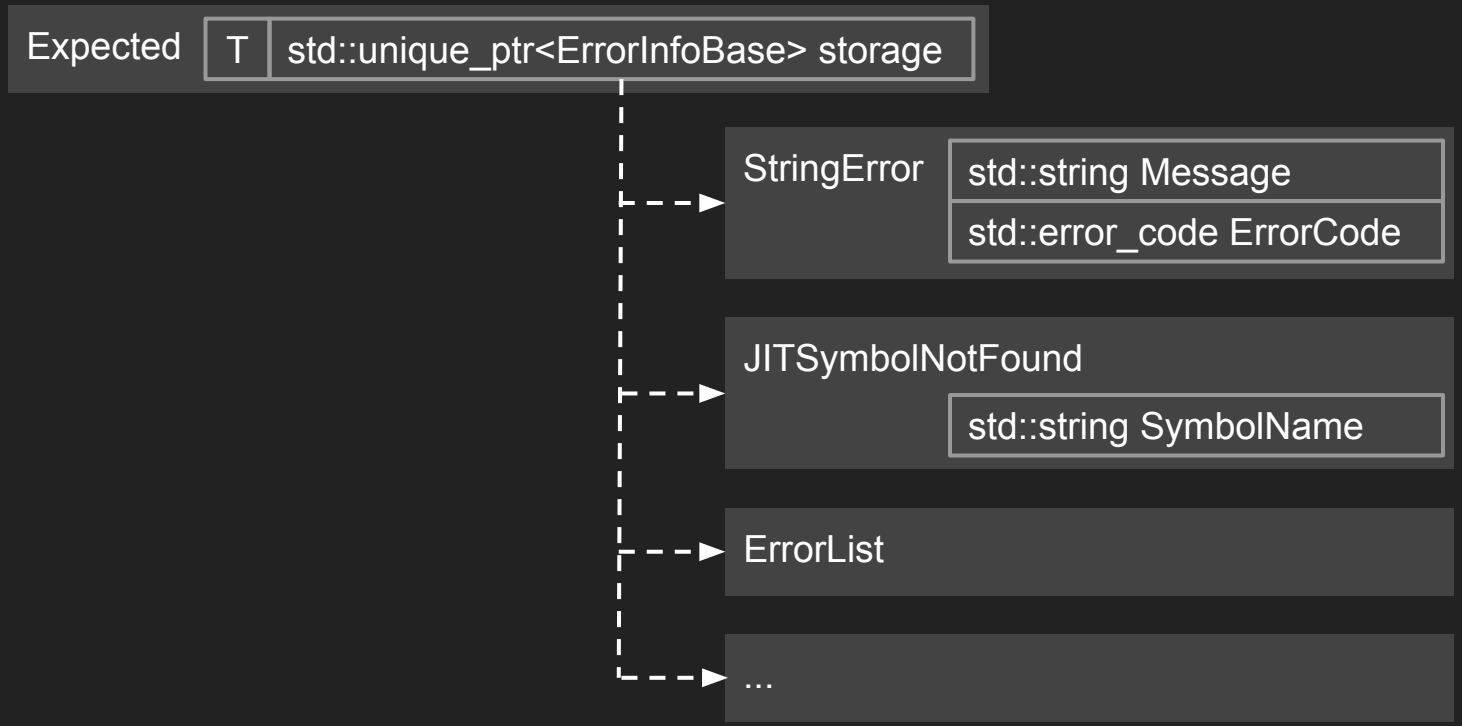
Composition



Composition



Composition



Utilities: make_error<T>

```
make_error<StringError>(
  "Bad executable",
  std::make_error_code(
    std::errc::executable_format_error));
```

Utilities: type-safe handlers

```
Error foo(...);

handleErrors(
    foo(...),
    [](const MyError &err){ ... },
    [](SomeOtherError &err){ ... });
```

Interop with std::error_code

```
std::error_code errorToErrorCode(Error err);
Error errorCodeToError(std::error_code ec);
```

- useful when for porting a codebase
- similar to *Exploding Return Codes*
  https://groups.google.com/forum/#!msg/comp.lang.c++.moderated/BkZqPfoq3ys/H_PMR8Sat4oJ

Example

```
bool simpleExample();

int main() {
  if (simpleExample())
    // ... more code ...

  return 0;
}
```

Example . expected

```
bool simpleExample() {
  std::string fileName = "[a*.txt";
  Expected<GlobPattern> pattern = GlobPattern::create(std::move(fileName));

  if (auto err = pattern.takeError()) {
    logAllUnhandledErrors(std::move(err), std::cerr, "[Glob Error] ");
    return false;
  }

  return pattern->match("...");
}
```

Output: `[Glob Error] invalid glob pattern: [a*.txt`

Example . error_code

```
bool simpleExample() {
  std::string fileName = "[a*.txt";
  GlobPattern pattern;

  if (std::error_code ec = GlobPattern::create(fileName, pattern)) {
    std::cerr << "[Glob Error] " << getErrorDescription(ec) << ": ";
    std::cerr << fileName << "\n";
    return false;
  }

  return pattern.match("...");
}
```

Output: `[Glob Error] invalid_argument: [a*.txt`

Example . modified

```
std::error_code simpleExample(bool &result, std::string &errorFileName) {
  GlobPattern pattern;
  std::string fileName = "[a*.txt";

  if (std::error_code ec = GlobPattern::create(fileName, pattern)) {
    errorFileName = fileName;
    return ec;
  }

  result = pattern.match("...");
  return std::error_code();
}
```

Example . clever

```
std::error_code simpleExample(bool &result, std::string *&errorFileName) {
  GlobPattern pattern;
  std::string fileName = "[a*.txt";

  if (std::error_code ec = GlobPattern::create(fileName, pattern)) {
    errorFileName = new std::string(fileName);
    return ec;
  }

  result = pattern.match("...");
  return std::error_code();
}
```

Example . modified

```
int main() {
  bool res;
  std::string *errorFileName = nullptr; // heap alloc in error case

  if (std::error_code ec = simpleExample(res, errorFileName)) {
    std::cerr << "[simpleExample Error] " << getErrorDescription(ec) << " ";
    std::cerr << *errorFileName << "\n";
    delete errorFileName;
    return 0;
  }

  // ... more code ...
  return 0;
}
```

Example . before

```
bool simpleExample() {
  std::string fileName = "[a*.txt";
  Expected<GlobPattern> pattern = GlobPattern::create(std::move(fileName));

  if (auto err = pattern.takeError()) {
    logAllUnhandledErrors(std::move(err), std::cerr, "[Glob Error] ");
    return false;
  }

  return pattern->match("...");
}
```

Example . after

```
Expected<bool> simpleExample() {
  std::string fileName = "[a*.txt";
  Expected<GlobPattern> pattern = GlobPattern::create(std::move(fileName));

  if (!pattern)
    return pattern.takeError();



  return pattern->match("...");
}
```

Example . after

```
int main() {
  Expected<bool> res = simpleExample();
  if (auto err = res.takeError()) {
    logAllUnhandledErrors(std::move(err), errs(), "[simpleExample Error] ");
    return 0;
  }

  // ... more code ...
  return 0;
}
```

Example . before

```
int main() {
  if (                    simpleExample()) {
    // ... more code ...


  }


  return 0;
}
```

Performance

- Concerned about NRVO when seeing code like this?

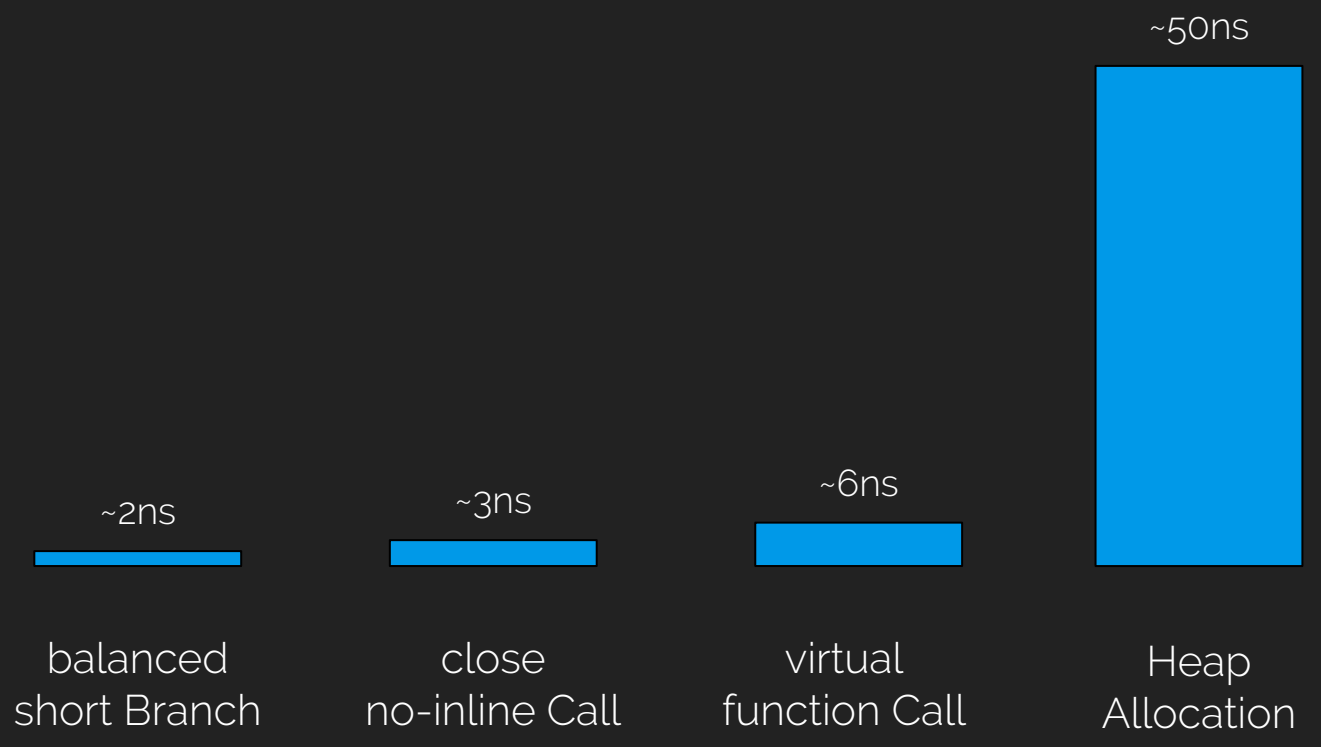
```
return std::move(error);
```

- Concerned about returning polymorphic objects?

  Instead of `bool`, `int`, `nullptr`, `std::error_code`

Yes, or course! We only pay for what we get!

Expected overhead category?



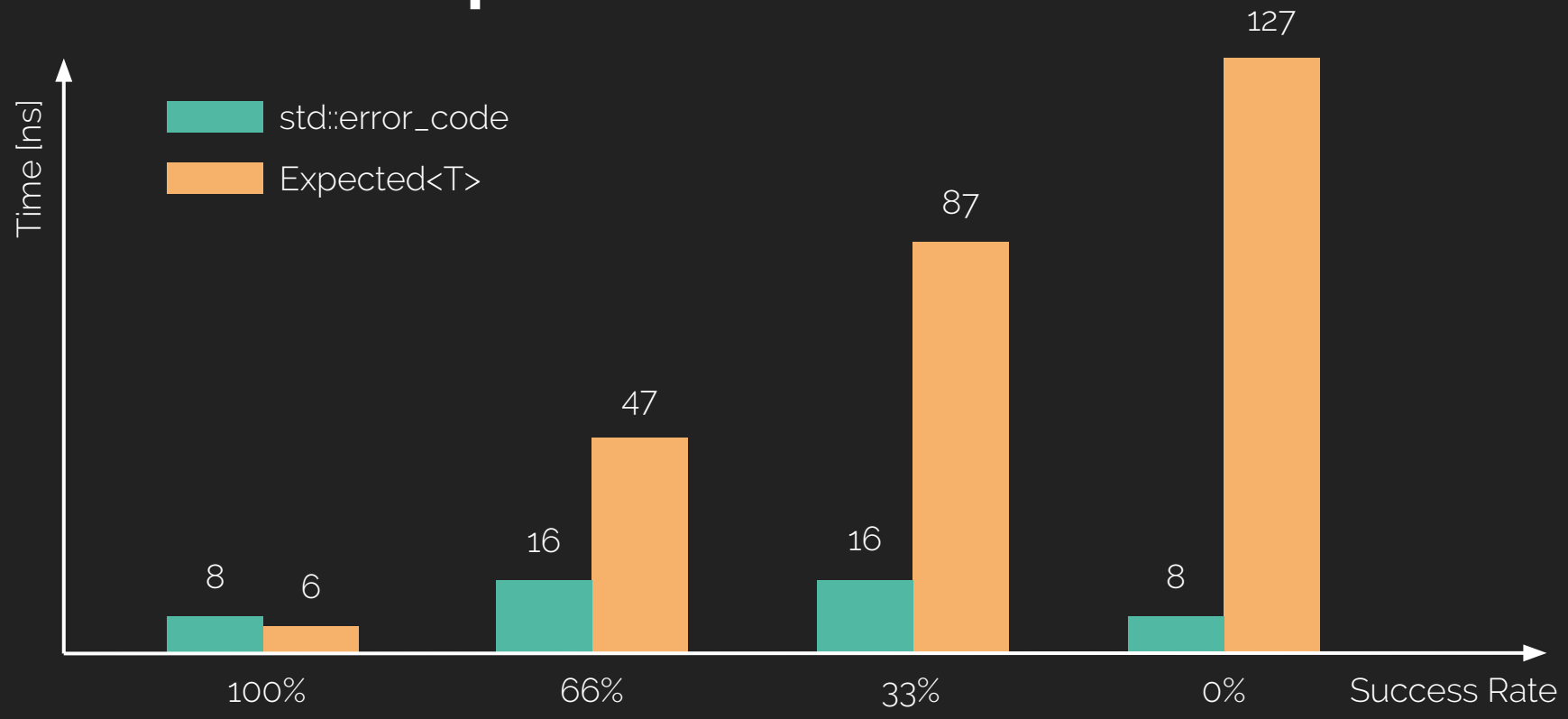
Minimal example . std::error_code

```
__attribute__((noinline))
static std::error_code Minimal_ErrorCode(int successRate, int &res) noexcept {
  if (fastrand() % 100 > successRate)
    return std::error_code(9, std::system_category());

  res = successRate;
  return std::error_code();
}
```

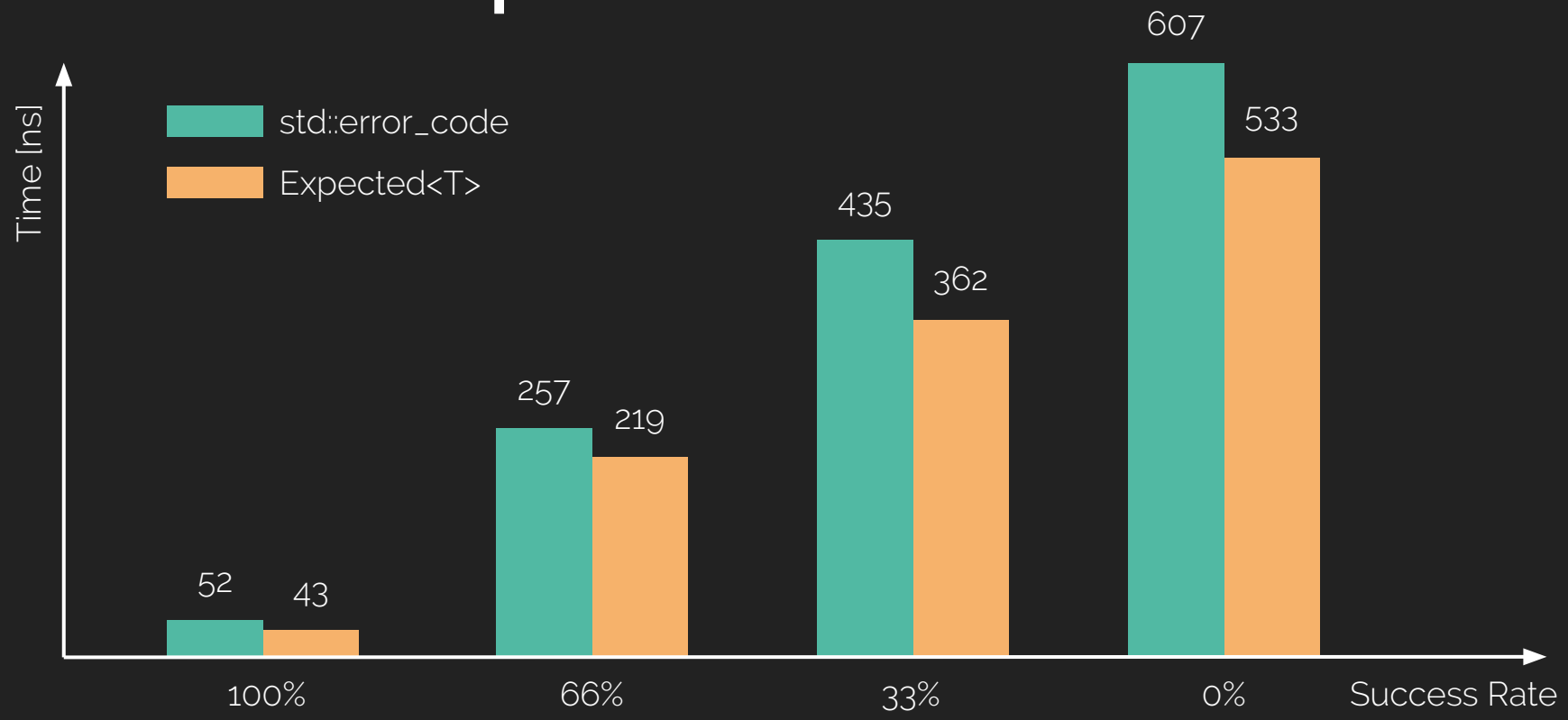
Minimal example . Expected<T>

```
__attribute__((noinline))
static llvm::Expected<int> Minimal_Expected(int successRate) noexcept {
  if (fastrand() % 100 > successRate)
    return llvm::make_error<llvm::StringError>(
        "Error Message", llvm::inconvertibleErrorCode());

  return successRate;
}
```

Minimal example



Previous example . after



Expected<T> vs. error code

- ✓ avoid vulnerabilities due to missed errors
- ✓ arbitrarily detailed error descriptions
- ✓ easily propagate errors up the stack
- ✓ no performance loss in success case

Differentiation

Alexandrescu's proposed `Expected<T>`

- made for interop with Exceptions (won't compile with `-fno-exceptions`)
- may pull in implementation-dependent trouble:

```
typedef /*unspecified*/ exception_ptr;
```

- supports Expected<void> where LLVM has Error

Differentiation

boost::outcome / std::experimental::expected

- interop with exceptions or error codes
- `expected<T, E>` has error type as template parameter
  - hard to build handy utilities around it
  - IMHO same mistake as static exception specifiers
    bad versionability, bad scalability: http://www.artima.com/intv/handcuffsP.html
- in progress, currently v2, maybe C++20

llvm::Expected<T> vs. others

- ✓ works in real code today
- ✓ supports error concatenation
- ✓ supports error type hierarchies
- ✓ great interop with std::error_code for converting APIs
- ✓ easy to understand, no unnecessary complexity
- ☐ not header-only

Test Idea

- Run a piece of code
- Count the number `N` of valid `Expected<T>` instances
- Execute the code `i = 1..N` times
- Turn the i'th valid instance into an error instance
- Each error path will be executed
- Potential issues show up
- Consider running with AddressSanitizer etc.

Dump Example

```
Expected<bool> simpleExample() {
  std::string fileName = "[a*.txt";
  Expected<GlobPattern> pattern = GlobPattern::create(std::move(fileName));

  if (pattern) // success case, frequently taken, good coverage
    return pattern->match("...");

  int x = *(int*)0; // runtime error, unlikely to show up in regular tests
  return pattern.takeError();
}
```

Naive Implementation

```
#ifndef NDEBUG
template <typename OtherT>
Expected(OtherT &&Val, typename std::enable_if<...>::type * = nullptr)
    : HasError(false), Unchecked(true)
{
  if (ForceAllErrors::TurnInstanceIntoError()) {
    HasError = true;
    new (getErrorStorage()) error_type(ForceAllErrors::mockError());
    return;
  }

  new (getStorage()) storage_type(std::forward<OtherT>(Val));
}
#else
...
```

Naive Testing

```
int breakInstance = 1..N;
ForceAllErrorsInScope FAE(breakInstance);

Expected<bool> expected = simpleExample();
EXPECT_FALSE(isInSuccessState(expected));

bool success = false;
handleAllErrors(expected.takeError(),
  [&](const ErrorInfoBase &err) { // no specific type information!
      success = true;
  });

EXPECT_TRUE(success);
```

Towards an Error Sanitizer

- Mock correct error type
  - extra info from static analysis → hack Clang
  - runtime support → extend & link LLVM Compiler-RT
- Support cascading errors
  - if error causes more errors, rerun and break all these too
- Avoid breaking instances multiple times
  - deduplicate according to `__FILE__` and `__LINE__`

Towards an Error Sanitizer

- Biggest challenge:
  Missed side effects can cause false-positive results

```
static int SideEffectValue = 0;

llvm::Expected<int> SideEffectExample(bool returnInt) {
  if (returnInt)
    return 0; // ESan breaks the instance created here

  SideEffectValue = 1; // regular errors include this side effect
  return llvm::make_error<CustomError>("Message");
}
```

Towards an Error Sanitizer

- Opinions welcome!
- More news maybe next year

Thks! Questions?

LLVM Programmer's Manual
http://llvm.org/docs/ProgrammersManual.html#recoverable-errors

Stripped-down Version of LLVM
https://github.com/weliveindetail/llvm-expected

Series of Blog Posts
http://weliveindetail.github.io/blog/

Naive Testing Implementation
https://github.com/weliveindetail/llvm-ForceAllErrors