Handling function failures

Andrzej Krzemieński

akrzemi1.wordpress.com

Failures

Disappointments in program:

- Disappointments in environment
- Bugs

Failures

```
vector<int> v (10); // 10 elemenets
v[10] = 0; // precondition violation -> NOT a failure
```

Failures

Failure:

- A function promised something
- And failed to do it

Failures

Failure:

- A function promised something
- And failed to do it
- Reflects disappointment in the environments

Handling failures

```
void communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        die("ERROR opening socket");

    hostent* server = gethostbyname(host);
    if (server == NULL)
      die("no such host");

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        die("ERROR connecting");

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
         die("ERROR writing to socket");

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
         die("ERROR reading from socket");

    printf("%s\n",buffer);
    close(sockfd);
}
```

Handling failures

```
void communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        die("ERROR opening socket");

    hostent* server = gethostbyname(host);
    if (server == NULL)
      die("no such host");

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        die("ERROR connecting");

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
         die("ERROR writing to socket");

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
         die("ERROR reading from socket");

    printf("%s\n",buffer);
    close(sockfd);
}
```

← open socket

Handling failures

```
void communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        die("ERROR opening socket");

    hostent* server = gethostbyname(host);
    if (server == NULL)
      die("no such host");

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        die("ERROR connecting");

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
         die("ERROR writing to socket");

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
         die("ERROR reading from socket");

    printf("%s\n",buffer);
    close(sockfd);
}
```

$\leftarrow$ identify server

Handling failures

```
void communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        die("ERROR opening socket");

    hostent* server = gethostbyname(host);
    if (server == NULL)
      die("no such host");

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        die("ERROR connecting");

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
         die("ERROR writing to socket");

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
         die("ERROR reading from socket");

    printf("%s\n",buffer);
    close(sockfd);
}
```

← establish connection

Handling failures

```
void communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        die("ERROR opening socket");

    hostent* server = gethostbyname(host);
    if (server == NULL)
      die("no such host");

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        die("ERROR connecting");

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
         die("ERROR writing to socket");

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
         die("ERROR reading from socket");

    printf("%s\n",buffer);
    close(sockfd);
}
```

← write to socket

Handling failures

```
void communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        die("ERROR opening socket");

    hostent* server = gethostbyname(host);
    if (server == NULL)
      die("no such host");

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        die("ERROR connecting");

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
         die("ERROR writing to socket");

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
         die("ERROR reading from socket");

    printf("%s\n",buffer);
    close(sockfd);
}
```

← read from socket

Handling failures

```
Errc communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        return Errc::open_socket_failed;

    hostent* server = gethostbyname(host);
    if (server == NULL)
        return Errc::no_such_host;

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        return Errc::connection_failed;

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
        return Errc::message_write_failed;

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
        return Errc::response_read_failed;

    printf("%s\n",buffer);
    close(sockfd);
    return Errc::Success;
}
```

Handling failures

```
Errc communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        return Errc::open_socket_failed;

    hostent* server = gethostbyname(host);
    if (server == NULL)
        return Errc::no_such_host;

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        return Errc::connection_failed;

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
        return Errc::message_write_failed;

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
        return Errc::response_read_failed;

    printf("%s\n",buffer);
    close(sockfd);
    return Errc::Success;
}
```

Handling failures

```
Errc communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        return Errc::open_socket_failed;

    hostent* server = gethostbyname(host);
    if (server == NULL)
        return Errc::no_such_host;

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        return Errc::connection_failed;

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
        return Errc::message_write_failed;

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
        return Errc::response_read_failed;

    printf("%s\n",buffer);
    close(sockfd);
    return Errc::Success;
}
```

Handling failures

```
Errc communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        return Errc::open_socket_failed;

    hostent* server = gethostbyname(host);
    if (server == NULL)
        return Errc::no_such_host;

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        return Errc::connection_failed;

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
        return Errc::message_write_failed;

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
        return Errc::response_read_failed;

    printf("%s\n",buffer);
    close(sockfd);
    return Errc::Success;
}
```

Handling failures

```
int doX(int x)
{
    int a = doA(x);

    int b = doB(a);

    return  doC(b);
}
```

Handling failures

```
int doX(int x)
{
    int a = doA(x);

    int b = doB(a);

    return  doC(b);
}
```

↑ *depends*

Handling failures

```
int doX(int x)
{
    int a = doA(x);
                      ↑ depends
    int b = doB(a);
                      ↑ depends
    return  doC(b);
}
```

Handling failures

```
int doX(int x)
{
    int a = doA(x);
                      ↑ depends
    int b = doB(a);
                      ↑ depends
    return  doC(b);
}
```

```
int f(int x)
{
    int i = doX(x);

    int j = doY(i);

    return  doZ(j);
}
```

Handling failures

```
int doX(int x)
{
    int a = doA(x);
                     ↑ depends
    int b = doB(a);
                     ↑ depends
    return  doC(b);
}
```

```
int f(int x)
{
    int i = doX(x);
                     ↑
    int j = doY(i);

    return  doZ(j);
}
```

Handling failures

```
int doX(int x)
{
    int a = doA(x);
                     ↑ depends
    int b = doB(a);
                     ↑ depends
    return  doC(b);
}
```

```
int f(int x)
{
    int i = doX(x);
                     ↑
    int j = doY(i);
                     ↑
    return  doZ(j);
}
```

Handling failures

Exceptions:

- Failure cascade
- Arbitrary data on failure
- Don’t waste return channel
- Separates failure logic
- Good default handling

Handling failures

Exceptions:

- Failure cascade
- Arbitrary data on failure
- Don’t waste return channel
- Separates failure logic
- Good default handling

Error codes:

- Minimum overhead
- Good worse case perf.
- Explicit failure paths
- Good for complex flows

Handling failures

Exception contract:

- Zero *run-time* overhead on success
- Potentially huge overhead on failure

Handling failures

Exception contract:

- Zero *run-time* overhead on success
- Potentially huge overhead on failure

→ Throw rarely, where overhead is irrelevant.

Handling failures

```
Errc communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        return Errc::open_socket_failed;

    hostent* server = gethostbyname(host);
    if (server == NULL)
        return Errc::no_such_host;

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        return Errc::connection_failed;

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
        return Errc::message_wtite_failed;

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
        return Errc::response_read_failed;

    printf("%s\n",buffer);
    close(sockfd);
    return Errc::Success;
}
```

Handling failures

```
Errc communicate(const char * host, int portno, const char * message)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        return Errc::open_socket_failed;

    hostent* server = gethostbyname(host);
    if (server == NULL)
        return Errc::no_such_host;

    sockaddr_in addr {AF_INET, htons(portno), {get_ip(*server)}};
    if (connect(sockfd, (sockaddr*) &addr, sizeof(addr)) < 0)
        return Errc::connection_failed;

    int n = write(sockfd, message, strlen(message));
    if (n < 0)
        return Errc::message_wtite_failed;

    char buffer[256] = {};
    n = read(sockfd,buffer,255);
    if (n < 0)
        return Errc::response_read_failed;

    printf("%s\n",buffer);
    close(sockfd);
    return Errc::Success;
}
```

Handling failures

```
auto r = acquire();
doA(r);
doB(r);
doC(r);
release(r); // not called
```

Handling failures

```
auto r = acquire();
doA(r);
doB(r);
doC(r);
release(r); // not called
```

Acquire and *if succeeds* schedule unconditional release on scope end.

Handling failures

```
Resource r {};
doA(r);
doB(r);
doC(r);
```

Acquire and *if succeeds* schedule unconditional release on scope end.

RAII – Resource Acquisition Is Initialization

Handling failures

```
Resource r {};
if (doA(r) == Err)
  return Err;
if (doB(r) == Err)
  return Err;
if (doC(r) == Err);
  return Err;
return 0;
```

Handling failures

```
Resource r {};
if (doA(r) == Err)
  return Err;
if (doB(r) == Err)
  return Err;
if (doC(r) == Err);
  return Err;
return 0;
```

It is *failure-safety*
(not exception safety)

Handling failures

Destructors used for:

- Resource release
- Final piece of business logic

Handling failures

```
int transfigure(int x, int y) {
  Resource res {};
  int z = compute(res, x, y);
  return z;
} // release
```

Handling failures

```
int transfigure(int x, int y) {
  Resource res {};
  int z = compute(res, x, y);
  return z;
} // release
```

- on failure cannot produce z

Handling failures

```
int transfigure(int x, int y) {
  Resource res {};
  int z = compute(res, x, y);
  return z;
} // release
```

- on failure cannot produce z

Handling failures

```
int transfigure(int x, int y) {
  Resource res {};
  int z = compute(res, x, y);
  return z;
} // release
```

- on failure we produced z

Handling failures

```
void save();
```

Handling failures

```cpp
void save() {
  std::ofstream f {"output.txt"};
  f << computeA();
  f << computeB();
} // flush in destructor
```

Handling failures

```
void save() {
  std::ofstream f; f.exceptions(failbit | badbit); f.open("out");
  f << computeA();
  f << computeB();
} // flush in destructor
```

Handling failures

```
void save() {
  std::ofstream f; f.exceptions(failbit | badbit); f.open("out");
  f << computeA();
  f << computeB();
} // flush in destructor
```

- reports failure to buffer

Handling failures

```
void save() {
  std::ofstream f; f.exceptions(failbit | badbit); f.open("out");
  f << computeA();
  f << computeB();
} // flush in destructor
```

- conceals failure to flush!

Handling failures

```
void save() {
  std::ofstream f; f.exceptions(failbit | badbit); f.open("out");
  f << computeA();
  f << computeB();
  f.flush();
} // only close in destructor
```

- reports failure to write

Handling failures

```
void save() {
  std::ofstream f; f.exceptions(failbit | badbit); f.open("out");
  f << computeA();
  f << computeB();
  f.flush();
} // only close in destructor
```

- saving cannot fail here

Error information

What information should an exception carry?

Error information

What information should an exception carry?

- Is catch site close or distant?

Error information

Local error handling:

```
try {
  fsys::copy_file(from, to);
}
catch (fsys::filesystem_error const& e) {
  log("failed to copy", e.path1(), e.path2());
}
```

Error information

Remote error handling:

```
for (;;) try {
  program_iteration();
}
catch (std::exception const& e) {
  log(e.what());
}
```

error_code

```
class error_code
{
  error_category* domain; // domain from which error originates
  int             value;  // value of error within domain
};
```

error_code

```
class error_code
{
  error_category* domain; // domain from which error originates
  int             value;  // value of error within domain
};
```

error_code

```
class error_code
{
  error_category* domain; // domain from which error originates
  int             value;  // value of error within domain
};
```

error_code

```
enum class ConvertErrc {
  StringTooLong = 1,
  EmptyString   = 2,
  IllegalChar   = 3,
};
```

error_code

```
enum class ConvertErrc {
  StringTooLong = 1,
  EmptyString   = 2,
  IllegalChar   = 3,
};

// plug ConvertErrc into error_code system
```

error_code

```
enum class ConvertErrc {
  StringTooLong = 1,
  EmptyString   = 2,
  IllegalChar   = 3,
};

// plug ConvertErrc into error_code system

std::error_code ec = ConvertErrc::EmptyString;
```

error_code

```
enum class ConvertErrc {
  StringTooLong = 1,
  EmptyString   = 2,
  IllegalChar   = 3,
};

// plug ConvertErrc into error_code system

std::error_code ec = ConvertErrc::EmptyString;
```

error_code

```
enum class ConvertErrc {
  StringTooLong = 1,
  EmptyString   = 2,
  IllegalChar   = 3,
};

// plug ConvertErrc into error_code system

std::error_code ec = ConvertErrc::EmptyString;
```

Error information

Boost.Exception

```
using from_path = boost::error_info<
  struct from_path_tag, // a tag type for uniqueness
  fsys::path >;         // type of the stored information
```

Error information

Boost.Exception

```
using from_path = boost::error_info<
  struct from_path_tag, // a tag type for uniqueness
  fsys::path >;         // type of the stored information
```

Error information

Boost.Exception

```
using from_path = boost::error_info<
  struct from_path_tag, // a tag type for uniqueness
  fsys::path >;         // type of the stored information
```

Error information

Boost.Exception

```
using from_path = boost::error_info<
  struct from_path_tag, // a tag type for uniqueness
  fsys::path >;         // type of the stored information

using to_path = boost::error_info<
  struct to_path_tag,
  fsys::path >;
```

Error information

Boost.Exception

```
using from_path = boost::error_info<
  struct from_path_tag, // a tag type for uniqueness
  fsys::path >;         // type of the stored information

using to_path = boost::error_info<
  struct to_path_tag,
  fsys::path >;
```

Error information

Boost.Exception

```
using from_path = boost::error_info<
  struct from_path_tag, // a tag type for uniqueness
  fsys::path >;         // type of the stored information

using to_path = boost::error_info<
  struct to_path_tag,
  fsys::path >;
```

Error information

Boost.Exception

```
catch (boost::exception & e) {
  e << from_path("input.txt");
  throw;
}
```

Error information

Boost.Exception

```
catch (boost::exception const& e) {
  if (const fsys::path * p = e.get_error_info<from_path>(e))
    log("from file", *p);
}
```

Error information

How many error types?

- For indicating different points at which to catch

When to throw?

Throw or not?

- `find()` on missing element?
- `convert()` on converting `"A" to int?`

When to throw?

Throw or not?

- Don’t throw if error expected to be handled locally
- Be prepared for unwinding the entire program

When to throw?

Throw or not?

- Don’t throw if error expected to be handled locally
- Be prepared for unwinding the entire program

- Would the advice be the same if throwing was cheap?

When to throw?

```
template <typename T>
T convert(std::string const& s);
```

When to throw?

```
config.port = default_value;
try {
  config.port = convert<int>(str);
}
catch (convert_error const&) {}
```

When to throw?

```
try {
  config.port = convert<int>(str);
}
catch (convert_error const&) {
  config.port = default_value;
}
```

When to throw?

```
try {
  config.port = convert<int>(str);
}
catch (convert_error const&) {
  throw MyProgramException{};
}
```

When to throw?

```
template <typename T>
bool convert(std::string const& s, T & v);
```

When to throw?

```
if (!convert(str, config.port))
  config.port = default_value;
```

When to throw?

```
if (!convert(str, config.port))
  throw MyProgramException{};
```

When to throw?

```
template <typename T>
[[nodiscard]] bool convert(std::string const& s, T & v);
```

When to throw?

```
convert(str, config.port); // warning → error
```

When to throw?

```
config.port = default_value;
convert(str, config.port); // warning → error
```

When to throw?

```
config.port = default_value;
(void)convert(str, config.port);
```

When to throw?

```
struct Success [[nodiscard]] { /* ... */ };
```

When to throw?

```
struct Success [[nodiscard]] { /* ... */ };



template <typename T>
// [[nodiscard]] inherited
Success convert(std::string const& s, T & v);
```

When to throw?

```
struct Success [[nodiscard]] { /* ... */ };



template <typename T>
// [[nodiscard]] inherited
Success convert(std::string const& s, T & v)
noexcept(false);
```

When to throw?

Two types of failure:

- Likely to be handled locally
- Likely to be handled remotely

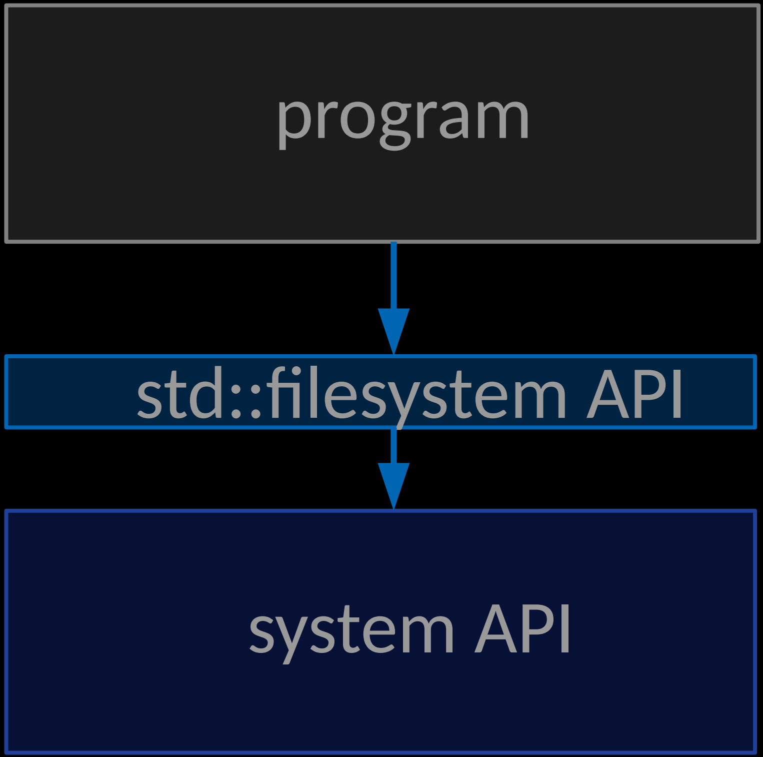
When to throw?

```
namespace std::filesystem
{
  uintmax_t file_size(const path& p);

  uintmax_t file_size(const path& p, error_code & ec);
}
```

When to throw?

```
namespace std::filesystem
{
  uintmax_t file_size(const path& p);

  uintmax_t file_size(const path& p, error_code & ec); // throws!
}
```

When to throw?



errors likely handled remotely

errors likely handled locally

When to throw?

```
namespace std::filesystem
{
  uintmax_t file_size(const path& p);

  uintmax_t file_size(const path& p, error_code & ec); // throws
}
```

Outcome

```
Success convert(std::string const& s, T & v);

uintmax_t file_size(const path& p, error_code & ec);
```

Outcome

Either a value or an error object!

Outcome

Either a value or an error object!

```
try {
  int i = convert<i>(str);              // either value
}
catch (conversion_error const& e) {  // or error
}
```

Outcome

Either a value or an error object!

```
try {
  int i = convert<i>(str);              // either value
}
catch (conversion_error const& e) {  // or error
}
```

Outcome

Either a value or an error object!

```
try {
  int i = convert<i>(str);              // either value
}
catch (conversion_error const& e) {  // or error
}
```

Outcome

```
template <typename T>
result<T> convert(string_view s);  // either T or std::error_code
```

Outcome

```
template <typename T>
result<T> convert(string_view s) { // either T or std::error_code
  if (any_failure)
    return ConvertErrc::TooLong;   // return error_code



}
```

Outcome

```
template <typename T>
result<T> convert(string_view s) { // either T or std::error_code
  if (any_failure)
    return ConvertErrc::TooLong;   // return error_code
  else
    return T{/*...*/};             // return T
}
```

Outcome

```
template <typename T>
result<T> convert(string_view s);  // either T or std::error_code



int i = convert<int>("S");         // will not compile
```

Outcome

```
template <typename T>
result<T> convert(string_view s);  // either T or std::error_code



int i = convert<int>("S").value(); // throws if no value
```

Outcome

```
template <typename T>
result<T> convert(string_view s);  // either T or std::error_code


if (result<int> r = convert<int>("S"))
  process_int(r.assume_value());
else
  report_error(r.assume_error());
```

Outcome

```
template <typename T>
result<T> convert(string_view s);  // either T or std::error_code


if (result<int> r = convert<int>("S"))
  process_int(r.value());
else
  report_error(r.error());
```

Outcome

```
template <typename T> result<T> convert(string_view s);
result<string> readFile(path const& p);


result<int> readInt(path const& p) {
  OUTCOME_TRY(str, readFile(p));
  OUTCOME_TRY(i, convert<int>(str));
  return i;
}
```

Outcome

```
template <typename T> result<T> convert(string_view s);
result<string> readFile(path const& p);
```

```
result<int> readInt(path const& p) {
  OUTCOME_TRY(str, readFile(p));
  OUTCOME_TRY(i, convert<int>(str));
  return i;
}
```

```
auto&& _r = readFile(p);
if (!_r) return _r.error();
auto&& str = _r.value();
```

Outcome

```
template <typename T> result<T> convert(string_view s);
result<string> readFile(path const& p);



result<int> readInt(path const& p) {
  string str = OUTCOME_TRYX(readFile(p));
  int i = OUTCOME_TRYX(convert<int>(str));
  return i;
}
```

Outcome

```
template <typename T> result<T> convert(string_view s);
result<string> readFile(path const& p);


result<int> readInt(path const& p) {
  return OUTCOME_TRYX(convert<int>(OUTCOME_TRYX(readFile(p))));
}
```

Outcome

```
result<void> validate(string_view s);



result<int> readInt(path const& p) {
  OUTCOME_TRY(str, readFile(p));
  validate(str);       // warning → error
  OUTCOME_TRY(i, convert<int>(str));
  return i;
}
```

Outcome

```
result<void> validate(string_view s);



result<int> readInt(path const& p) {
  OUTCOME_TRY(str, readFile(p));
  OUTCOME_TRY((validate(str)));
  OUTCOME_TRY(i, convert<int>(str));
  return i;
}
```

Outcome

```
result<void> validate(string_view s);



result<int> readInt(path const& p) {
  OUTCOME_TRY(str, readFile(p));
  OUTCOME_TRY((validate(str)));
  OUTCOME_TRY(i, convert<int>(str));
  return i;
}
```

```
auto&& _r = validate(str);
if (!_r) return _r.error();
```

Outcome

Three choices:

• handle manually	`if (auto r = f())`
• lightweight “throw”	`OUTCOME_TRY(f())`
• change to normal throw	`f().value()`

Outcome

Factories

```
class File {
private: explicit File(FILE* handle);
         File(File&&); // no copying
         FILE* handle_;

public:  static result<File> create(const char* p);
};
```

Outcome

Factories

```
class File {
private: explicit File(FILE* handle);
         File(File&&); // no copying
         FILE* handle_;

public:  static result<File> create(const char* p);
};
```

Outcome

Factories

```
class File {
private: explicit File(FILE* handle);
         File(File&&); // no copying
         FILE* handle_;

public:  static result<File> create(const char* p);
};
```

Outcome

Factories

```
class File {
private: explicit File(FILE* handle);
         File(File&&); // no copying
         FILE* handle_;

public:  static result<File> create(const char* p);
};
```

Outcome

Factories

```
result<File> File::create(const char* p) {
  if (FILE* h = fopen(p, "r"))
    return result<File>{in_place_type<File>, h};
  else
    return FileErrc::OpenFailed;
}
```

Outcome

Factories

```
std::mutex create() {
  return std::mutex{}; // no copy or move
}
```

Outcome

Factories

```
OUTCOME_TRY(f, File::create("output.txt"));
OUTCOME_TRY(s, f.readLine());
```

Throw by value

Throw by value

```
int convert(string_view s) throws;  // either return int or
                                    // return-throw std::error
```

Throw by value

```
int convert(string_view s) throws {
  if (s.empty())
    throw ConvErrc::Empty;          // return-throw std::error
  // ...
}
```

Throw by value

```
try {
  int i = convert(str);
  use(i);
}
catch(std::error e) {          // zero-overhead
  inspect(e);
}
```

Throw by value

```
try {                                    //  if (auto r = convert(str))
  int i = convert(str);                  //    use(r.value());
  use(i);                                //
}                                        //  else
catch(std::error e) {                    //    inspect(r.error());
  inspect(e);
}
```

Throw by value

```
try {                                 //  if (auto r = convert(str))
  int i = convert(str);               //    use(r.value());
  use(i);                             //
}                                     //  else
catch(std::error e) {                 //    inspect(r.error());
  inspect(e);
}
```

Throw by value

```
try {                                    //  if (auto r = convert(str))
  int i = convert(str);                  //    use(r.value());
  use(i);                                //
}                                        //  else
catch(std::error e) {                    //    inspect(r.error());
  inspect(e);
}
```

Designing failure handling

Designing failure handling

```
{"request": {
  "from": "FRA",
  "to"   : "LHR",
  "date": "2018-11-07"
}}
```

Designing failure handling

```
{"response": [
  {"flight": "LH1011", "departs": "10:45", "arrives": "12:10"},
  {"flight": "BA6061", "departs": "07:00", "arrives": "08:20"},
  {"flight": "BA6063", "departs": "17:00", "arrives": "18:20"}
]}
```

Designing failure handling

```
{"response" : {
  "error"   : "REQ.102"
  "message" : "no such airport"
}}
```

Designing failure handling

```
{"response" : {
  "error" : "java.lang.NullPointerException: null
                at.commycomp.server.app.servlet.Compute
                at javax.servlet.http.HttpServlet.service
                at org.apache.coyote.AbstractProtocol
                at org.apache.tomcat.util.net.NioEndpoint.run
                at java.util.concurrent.ThreadPoolExecutor
                at java.lang.Thread.run"
}}
```

Handling failures

- Instruction dependency
- Good communication

Handling failures

- Instruction dependency
- Good communication
- Choice of tools is up to you!