Error handling is cancelling operations

Andrzej Krzemieński

akrzemi1.wordpress.com

Motivation

People get incorrect picture of exceptions:

- They cause memory leaks
- They cause your program to crash
- “Catch them as soon as possible, then think”

Motivation

Provide a model for describing the handling of function failures.

Failures vs bugs

Error:

- Making a wrong choice
- Deviation from truth
- A mistake

Failures vs bugs

Bug:

Program does something else than what programmer intended.

Failure:

program reports a “disappointment” exactly as programmer intended.

Failures vs bugs

Bug:

- Caused by programmer
- Correct response is to change code
- Detected by static analysis
- Different tool for handling needed
  - Type-system
  - Contracts

```
double b = any_val();
double d = x*x + y*y;
double dist = sqrt(b);
```

Failures vs bugs

Bug:

- Caused by programmer
- Correct response is to change code
- Detected by static analysis
- Different tool for handling needed
  - Type-system
  - Contracts

```
double b = any_val();
double d = x*x + y*y;
double dist = sqrt(b);
// should be `sqrt(d)`
```

Failures vs bugs

Failure:

- Caused by environment, unpredictable statically
- Correct response is to take different branch in code

Failures vs bugs

```
int open_socket();
```

Contract:

- *either* opens a socket (postcondition),
- *or* reports failure.

Failures vs bugs

```
int open_socket();
```

Contract:

- *either* opens a socket (postcondition),
- *or* reports failure.

- Implies branches in the caller.

Success dependency

Success dependency

```
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);
}
```

Success dependency

```
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);
}
```

Success dependency

```
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

Success dependency

```
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);
}
```

← resolve server hostname

Success dependency

```
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);
}
```

← connect to server

Success dependency

```
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);
}
```

← send data

Success dependency

```
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);
}
```

← receive data

Success dependency

```
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);
}
```

$\longleftarrow$ output data

Success dependency

```
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);
}
```

Success dependency

```
open_socket();
if (failed) die();

resolve_host();
if (failed) die();

connect();
if (failed) die();

send_data();
if (failed) die();

receive_data();
if (failed) die();
```

Success dependency

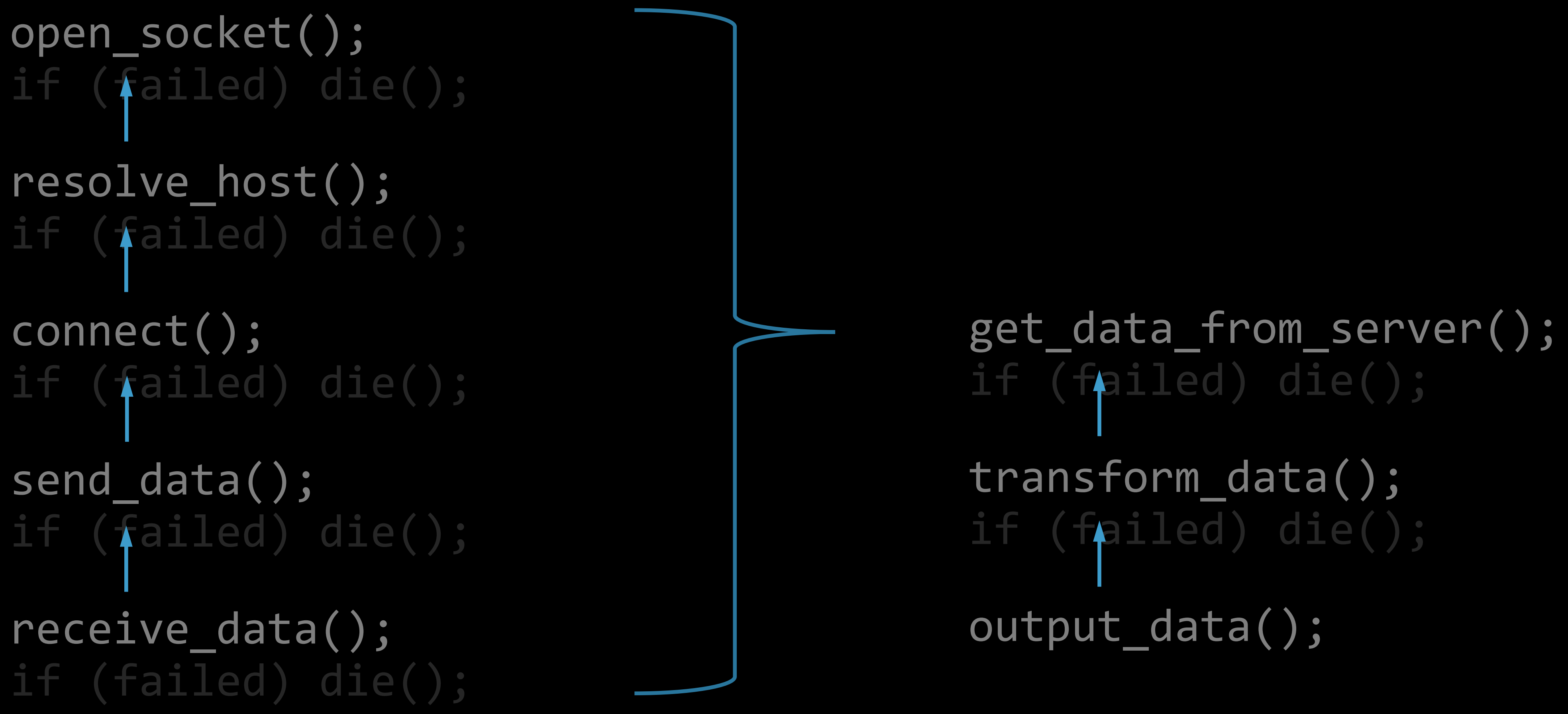
```
open_socket();
if (failed) die();

resolve_host();
if (failed) die();

connect();
if (failed) die();

send_data();
if (failed) die();

receive_data();
if (failed) die();
```

Success dependency

```
open_socket();
if (failed) die();

resolve_host();
if (failed) die();

connect();
if (failed) die();

send_data();
if (failed) die();

receive_data();
if (failed) die();
```

- bug if we call `resolve_host()` when `open_socket()` has failed
- `resolve_host()`’s postcondition is `open_socket()`’s precondition

Success dependency



```
open_socket();
if (failed) die();

resolve_host();
if (failed) die();

connect();
if (failed) die();

send_data();
if (failed) die();

receive_data();
if (failed) die();
```

```
get_data_from_server();
if (failed) die();

transform_data();
if (failed) die();

output_data();
```

Success dependency

```
open_socket();
if (failed) die();

connect();
if (failed) die();

obtain_data();
if (failed) die();
```

Success dependency

```
open_socket();
if (failed) die();

connect();
if (failed) die();

obtain_data();
if (failed) die();
```

Do I have to `die()` on any failure?

Success dependency

Breaking the cancellation cascade:

- When next operation doesn’t mind if the previous one fails.
- Not many such situations.

Success dependency

Breaking the cancellation cascade:

- We require data from at least one of three servers.
- Processing next request in the server.

Success dependency

```
while (server_is_up) {
  const Request rq = queue.pop();
  try {
    process(rq);
  }
  catch (exception const& e) {
    report_failure(rq, e);
  }
}
```

- Next iteration does not depend on the previous

Success dependency

```
Status get_data_from_server() {
  open_socket();
  if (failed) return failure();

  connect();
  if (failed) return failure();

  obtain_data();
  if (failed) return failure();

  return success();
}
```

Success dependency

```
Status get_data_from_server() {
  open_socket();
  if (failed) return failure();

  connect();
  if (failed) return failure();

  obtain_data();
  if (failed) return failure();

  return success();
}
```

Exceptions for ergonomy

```
void get_data_from_server() {
  open_socket();
  connect();
  obtain_data();
}
```

Success dependency

```
Status get_data_from_server() {
  open_socket();
  // if (failed) return failure();

  connect();
  if (failed) return failure();

  obtain_data();
  if (failed) return failure();

  return success();
}
```

Safety issues

Success dependency

```
Status get_data_from_server() {
  open_socket();
  // <-- compiler warning

  connect();
  if (failed) return failure();

  obtain_data();
  if (failed) return failure();

  return success();
}
```

Safety issues

```
class Status [[nodiscard]] {
  // ...
}
```

Handling resources

Handling resources

```
Status get_data_from_server() {
  open_socket();
  if (failed) return failure();

  connect();
  if (failed) return failure();

  obtain_data();
  if (failed) return failure();

  close_socket();
  return success();
}
```

Handling resources

```
Status get_data_from_server() {
  open_socket();
  if (failed) return failure();

  connect();
  if (failed) return failure();

  obtain_data();                  // <-- if this fails
  if (failed) return failure();

  close_socket();                 // <-- this is skipped
  return success();
}
```

Handling resources

Resource:

- Something we need to share with others

Session:

- Since we successfully *acquired* the resource
- Until we *release* the resource.

Handling resources

Dependencies with resources:

- Resource release depends on resource acquisition
- Resource usage depends on resource acquisition
- Resource release does *not* depend on resource usage

$$\begin{tikzcd}
\texttt{open\_socket();} \\
\texttt{connect();} \arrow[u] \\
\texttt{obtain\_data();} \arrow[u] \\
\texttt{close\_socket();} \arrow[u, dotted, "/" description] \arrow[uuu, bend right]
\end{tikzcd}$$

Handling resources

Object lifetime represents a Resource Session:

- Constructor acquires the resource
- Destructor releases the resource

```
Status get_data_from_server() {
  Socket socket {};
  if (failed) return failure();

  connect();
  if (failed) return failure();

  obtain_data();
  if (failed) return failure();

  return success();
} // <-- socket closed here
```

Handling resources

Object lifetime represents a Resource Session:

- Constructor acquires the resource
- Destructor releases the resource
- This is called RAII

```
Status get_data_from_server() {
  Socket socket {};
  if (failed) return failure();

  connect();
  if (failed) return failure();

  obtain_data();
  if (failed) return failure();

  return success();
} // <-- socket closed here
```

Handling resources

Can we fail to close the socket?

Handling resources

Can we fail to close the socket?

- Yes.
- Do our callers care?

Handling resources

- Resources are means, not the goal.
- Function’s contract is never to manage a resource.
- Callers depend on the declared contract – not managing resources.

Handling resources

```
Status get_data_from_server() {
  Socket socket {};
  if (failed) return failure();

  connect();
  if (failed) return failure();

  obtain_data();
  if (failed) return failure();

  return success();
} // <-- socket closed here
```

← if fails, no data

Handling resources

```
Status get_data_from_server() {
  Socket socket {};
  if (failed) return failure();

  connect();
  if (failed) return failure();

  obtain_data();
  if (failed) return failure();

  return success();
} // <-- socket closed here
```

← if fails, no data

Handling resources

```
Status get_data_from_server() {
  Socket socket {};
  if (failed) return failure();

  connect();
  if (failed) return failure();

  obtain_data();
  if (failed) return failure();

  return success();
} // <-- socket closed here
```

← if fails, no data

Handling resources

```
Status get_data_from_server() {
  Socket socket {};
  if (failed) return failure();

  connect();
  if (failed) return failure();

  obtain_data();
  if (failed) return failure();

  return success();
} // <-- socket closed here
```

← if fails, we have data!

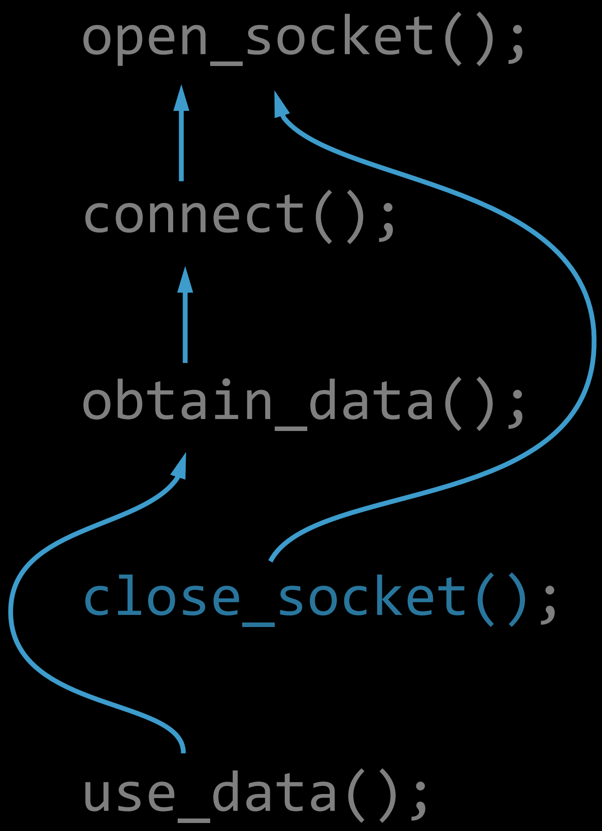
Handling resources

- We *do not* throw to say that something failed!
- We throw to say that something must be canceled.

Handling resources

Dependencies with resources:

- Resource release depends on resource acquisition
- Resource usage depends on resource acquisition
- Resource release does *not* depend on resource usage
- Subsequent operations do not depend on resource release

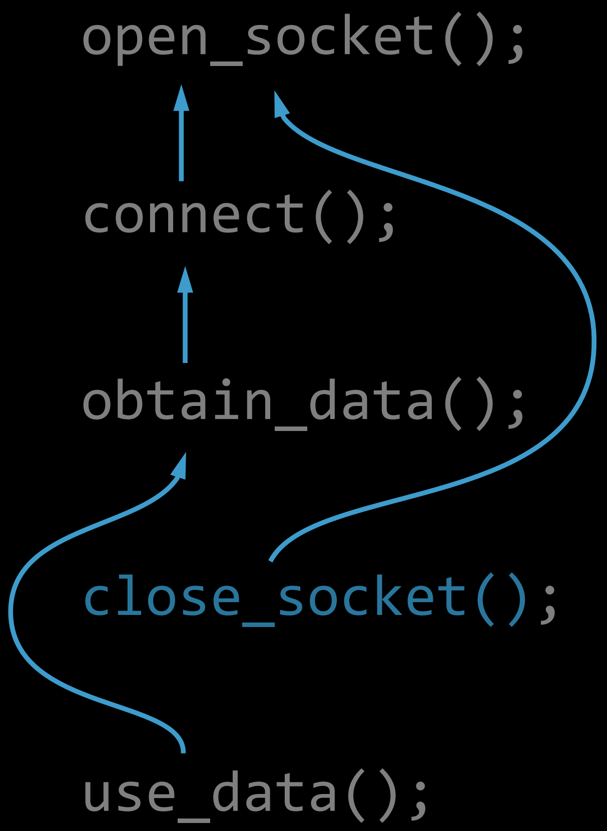


```
open_socket();

connect();

obtain_data();

close_socket();

use_data();
```

Handling resources

Reason for not throwing from destructors:

- No success dependency
- Fear for “double exception” is irrelevant

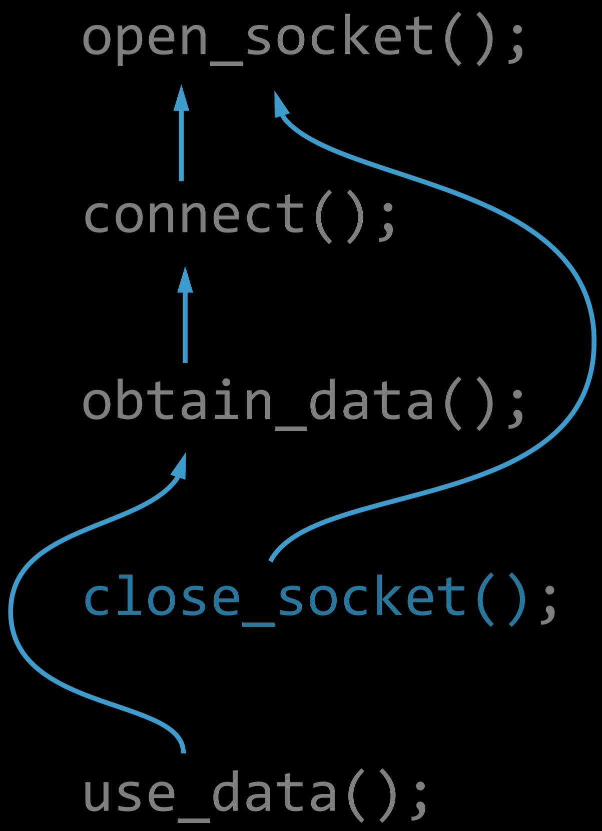


```
open_socket();

connect();

obtain_data();

close_socket();

use_data();
```

Handling resources

Identify resources in your program

- Does it match dependency patterns of resources?
- If so, manage resource sessions with constructors & destructors



```
open_socket();

connect();

obtain_data();

close_socket();

use_data();
```

Handling resources

Can I use destructors for other things?

Handling resources

Can I use destructors for other things?

- Yes, but...
- No language support
- No easy model

Handling resources

Can I use destructors for other things?

- Yes, but…
- No language support
- No easy model

We need:

$$\begin{tikzcd}
\texttt{create\_rec();} \\
\texttt{fill\_rec();} \arrow[u] \\
\texttt{commit\_rec();} \arrow[u] \\
\texttt{use\_rec();} \arrow[u]
\end{tikzcd}$$

We get:

$$\begin{tikzcd}
\texttt{create\_rec();} \\
\texttt{fill\_rec();} \arrow[u] \\
\texttt{commit\_rec();} \arrow[u] \arrow[uu, bend right] \\
\texttt{use\_rec();} \arrow[u]
\end{tikzcd}$$

Handling resources

Can I use destructors for other things?

- Yes, but…
- No language support
- No easy model

We need:

$$\begin{tikzcd}
\texttt{create\_rec();} \\
\texttt{fill\_rec();} \arrow[u] \\
\texttt{commit\_rec();} \arrow[u] \\
\texttt{use\_rec();} \arrow[u]
\end{tikzcd}$$

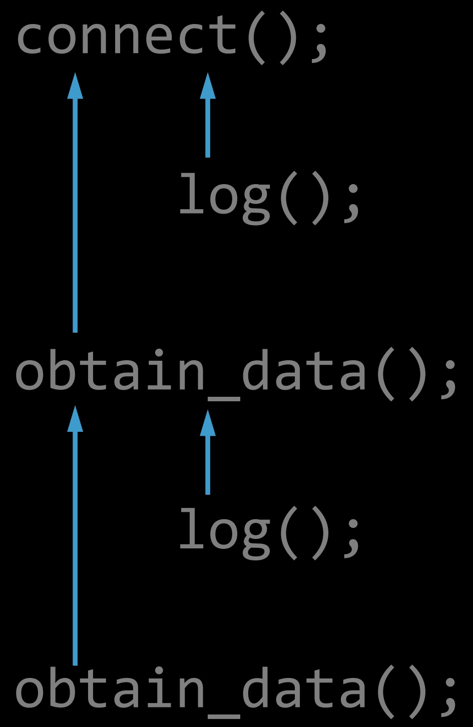
We get:

$$\begin{tikzcd}
\texttt{create\_rec();} \\
\texttt{fill\_rec();} \arrow[u] \\
\texttt{commit\_rec();} \arrow[u] \arrow[uu, bend right] \\
\texttt{use\_rec();} \arrow[u]
\end{tikzcd}$$

Logging

Logging

```
resolve_host();
if (failed) return failure();

log();

connect();
if (failed) return failure();

log();

obtain_data();
if (failed) return failure();
```



Basic failure safety

Basic failure safety

Cancellation cascade

- Unwinds scopes
- Destroys objects in scopes
- Ideally, does nothing else

After any failed operation

- Must be possible to *safely* destroy object
- Guaranteeing any specific state often not necessary

Basic failure safety

```
class Person {
  string first_name;
  string last_name;

  Person& Person::operator=(Person const& p) {
    first_name = p.first_name;
    last_name = p.last_name;
    return *this;
  }
};
```

Basic failure safety

```
class Person {
  string first_name;
  string last_name;

  Person& Person::operator=(Person const& p) {
    first_name = p.first_name;
    last_name = p.last_name; // <-- if fails
    return *this;
  }
};
```

Basic failure safety

```
void process(Person & p);
  // precondition: p is existing Person

void fun () {
  Person p = next_person();
  process(p);
  p = next_person();
  process(p);
};
```

Basic failure safety

```
void process(Person & p);
  // precondition: p is existing Person

void fun () {
  Person p = next_person();
  process(p);
  p = next_person(); // <-- throws: bad Person created
  process(p);
};
```

Basic failure safety

```
void process(Person & p);
  // precondition: p is existing Person

void fun () {
  Person p = next_person();
  process(p);
  p = next_person(); // <-- throws: bad Person created
  process(p);        // <-- potential problem skipped
};
```

Basic failure safety

```
void process(Person & p);
  // precondition: p is existing Person

void fun () {
  Person p = next_person();
  process(p);
  p = next_person(); // <-- throws: bad Person created
  process(p);        // <-- potential problem skipped
};                   // <-- bad Person safely destroyed
```

Basic failure safety

```
Person p = next_person();
try {
  p = next_person();
}
catch(...) {
  // TODO: handle it
}
process(p);           // <-- bad Person potentially observed
```

Basic failure safety

```
Person p = next_person();
try {
  p = next_person();
}
catch(...) {          // <-- cascade stopped prematurely
  // TODO: handle it
}
process(p);           // <-- bad Person potentially observed
```

Basic failure safety

```
while (server_is_up) {
  const Request rq = queue.pop();
  try {
    process(rq);
  }
  catch (exception const& e) {
    report_failure(rq, e);
  }
}
```

- Special case
- Stronger guarantee required

Basic failure safety

```
while (server_is_up) {
  const Request rq = queue.pop();
  try {
    process(rq);
  }
  catch (exception const& e) {
    report_failure(rq, e);
  }
}
```

- Const object
- Cannot be set to bad state
- Failure safety guaranteed

Basic failure safety

```
{
  Person p = next_person();
  Scope_guard _ = [&]{ process(p); };
  p = next_person();
} // process(p);     // <-- bad Person potentially observed
```

Basic failure safety

```
{
  Person p = next_person();
  Scope_guard _ = [&]{ process(p); }
  p = next_person();
} // process(p);     // <-- bad Person potentially observed
                     //     destructor doing more than
                     //     releasing resources
```

Basic failure safety

When any function fails:

- No resources are leaked

When member function fails:

- Object can be safely destroyed

Basic failure safety

When any function fails:

- No resources are leaked

When member function fails:

- Object can be safely destroyed
- Object’s invariants are preserved
  - *Valid* but unspecified state
  - Any operation without preconditions can be *safely* invoked

Basic failure safety

Basic failure safety:

- Applies to any failure-handling technique
- Enables the cancellation cascade to work correctly

Usable information

Usable information

```
while (server_is_up) {
  const Request rq = queue.pop();
  try {
    process(rq);
  }
  catch (exception const& e) {
    report_failure(rq, e);
  }
}
```

- What info from `e` can we effectively use?

Usable information

Generic use of exceptions:

- Only need failure code (`lib-A:101`)
- Human-readable text
- Indicate at which level to stop the cancellation cascade
- No exception type hierarchy required!

Using the model

Using the model

Q: Should I catch the exception?

A: Do subsequent operations depend on the one that failed?

Using the model

Q: Should I throw an exception?

A: Do callers need subsequent operations need to be canceled?

Using the model

Q: Should I throw from destructor?

A: Using a destructor for something else than releasing resources?

- Do you understand all the gotchas?
- Throwing makes sense when you emulate a straight success dependency path

Using the model

Q: What if nobody catches my exception and the program terminates?

A: Either all subsequent instructions depend on yours

- In that case all of the program needs to be cancelled

Or there is a bug in the caller that fails to stop the cascade

- In that case the caller needs a fix,
- and you need to throw

Using the model

Q: What if my caller is not “exception-safe”?

A: What do you mean by “exception-safe”?

- Not observing the cancellation cascade?
  - There is a more serious problem to be fixed in the caller
- Caller is using a different failure-handling technique?
  - Indeed, you need to adapt: maybe catch and translate to status code

Using the model

Q: What if throwing an exception incurs unacceptable cost?

A: Sometimes it is an unjustified belief

Sometimes it is really the case:

- Exceptions have certain performance trade-offs
- They may not work in your domain

Failure handling – summary

- Observe success dependency
  - Do not stop cancellation cascade prematurely
- Observe resource handling patterns
  - Identify your resources
  - Use destructors only for releasing resources, don’t throw
- Observe basic failure safety
- This applies to *any* failure handling technique.