# C++ Exceptions

Written by:

- Bernard Teo Zhi Yi

Last updated: 7 July 2022



## Feedback form for Lecture 7

Please fill in the feedback form for lecture 7.

# 1 Error handling mechanisms

In most large programs, there are many ways in which a function can fail. For example, you might have:

- A function that parses an string representing a binary number, but you give it `"10011210"`
- A function that expects a duration as a `double`, but you give it a NaN value
- A function that opens a TCP connection, but your machine is not connected to the Internet
- A function that deletes a file, but the file doesn't exist
- A function that needs to allocate a large memory buffer, but not enough memory is available

In all these examples, these functions are unable to perform their intended tasks due to situations that are out of their control.

How should we handle such errors?

C APIs typically use some kind of error code, typically seen in system calls:

```
char buf[64];
ssize_t res = read(fd, buf, 64);
if (res < 0) {
  printf("read() returned an error: %s", strerror(errno));
} else {
  /* parse the buffer */
}
```

Snippet 1: `read` system call

We could also write such functions that either set `errno` or return some kind of error code, but the situation quickly gets unwieldy when we need to pass the error out multiple layers of function calls.

```
const int SUCCESS = 0;
const int FILE_DOES_NOT_EXIST = 1;
const int FILE_CANNOT_BE_READ = 2;
const int CANNOT_PARSE_AS_INT = 3;
const int INT_TOO_LARGE = 4;
int readAnIntFromFile(const char* filename, int& out) {
  int fd = open(filename, O_RDONLY);
  if (fd == -1) {
    return FILE_DOES_NOT_EXIST;
  }
  char buf[64];
  ssize_t len = read(fd, buf, 64);
  if (len < 0) {
    return FILE_CANNOT_BE_READ;
  }
  close(fd);
  int val;
  auto [end, ec] = std::from_chars(buf, buf + len, val);
  if (ec != std::errc{}) {
    if (ec == std::errc::result_out_of_range{}) {
      return INT_TOO_LARGE;
    }
    return CANNOT_PARSE_AS_INT;
  }
  out = val;
  return SUCCESS;
}
```

Snippet 2: Wrapper for reading an int

Handling the error conditions makes the code a lot longer.

And there are still many further issues with this function:

- If `read` returns an error, the file will be left unclosed
- It leaks an implementation detail to the caller — the caller needs to know that it can check `errno` if they get the error code of `FILE_DOES_NOT_EXIST` or `FILE_CANNOT_BE_READ`, but not for the other two errors
- We're not checking if `close` succeeds

And the mess doesn't end here — we didn't actually do anything to "fix" the errors in this function. Instead, we propagated those errors out to the caller, which will need to handle them as well.

It would be much nicer if we were able to write code that describes just the non-exceptional case (i.e. the "good" case), and only check for the errors at the point where we can handle them (e.g. by checking for a different file, or by showing an alert to the user). We would then be able to write the same function much more concisely (using the hypothetical calls `open_ex`, `read_ex`, and `atoi_ex`):

```
int readAnIntFromFile_ex(const char* filename) {
  FileHandle hdl = open_ex(filename, O_RDONLY);
  char buf[64];
  size_t len = read_ex(hdl, buf, 64);
  return atoi_ex(buf, len);
}
```

Snippet 3: Wrapper for reading an int, without explicit error handling

Then we could perhaps handle the error conditions in the caller:

```
int doCommand() {
  int tries = 0;
  char filename[64] = "/path/to/my/file";
  size_t len = strlen(filename);
  while (true) {
    itoa(tries, filename + len, 10);
    try {
      int val = readAnIntFromFile_ex(filename);
      // success! lets do stuff with val
      return doStuff(val);
    } catch (const FileNotFoundException& e) {
      // too bad, can't find file
      showAlertMessage("Cannot find a suitable file");
      return;
    } catch (...) { // catch all other exceptions
      // try again with next file
      ++tries;
    }
  }
}
```

Snippet 4: Actually handling the error conditions

With appropriately written `open_ex`, `read_ex`, and `atoi_ex`, we would be able to write code like the above, where all the code for handling those error conditions only need to be written where we are able to handle those errors (e.g. by retrying or warning the user). Importantly, there may be arbitrarily many stack frames between the error handlers and the function call that actually causes the error, and the errors must still be propagated out "automagically".

This style of error handling is known as **exceptions**, and they are a feature of many commonly-used programming languages. The code that raises the error (e.g. in `atoi_ex`) is said to **throw** an exception, and the handlers (i.e. the `catch` blocks above) are said to **catch** them — perhaps called as such since control flow abruptly jumps to code seemingly far away from the throw site.

## 1.1 Concerns when designing an exception handling language feature

There are various concerns when designing an exception handling in a programming language, and various languages have taken slightly different approaches.

Some of the more important concerns are:

- **Release of resources**. Resources are often acquired during normal execution of a program, and they are usually characterised by having a function to acquire the resource and another function to release it (recall the lecture on ownership). The most common resource is heap memory (`new` / `delete` or `malloc` / `free`), but there are various other resources, including file handles (`open` / `close` or `fopen` / `fclose`), mutexes (`pthread_mutex_init` / `pthread_mutex_destroy`), and locks (`pthread_mutex_lock` / `pthread_mutex_unlock`). While an exception is thrown out of a stack frame, we want all resources to be released, so that the code does not leak those resources — our first example in this lecture shows how easy it is to accidentally leak resources when an error occurs.
- **Error descriptions and additional data**. We often want to pass additional information about the error. In the simplest cases, an error code might suffice, but might want to provide additional information which the exception handler might use. Even if the exception handler is unable to recover, we might want to provide a string that the handler can display to the user.
- **Ensuring that all exception types are handled**. As we see from the previous example, we often need to handle multiple different types of exceptions, as exceptions may arise from different situations. It is desirable for the language to be able to detect if the user has forgotten to handle some exception types.
- **Efficiency**. What is the cost (i.e. time or CPU cycles) of throwing and propagating an exception? How much additional overhead did we introduce if the code took the non-exceptional code path? This is usually a tradeoff, but if exceptions are only thrown in *exceptional* situations (i.e. extremely rarely, and not during normal execution of the program), then we should focus on minimising the overhead in the non-exceptional code path.

Keep these concerns in mind as we go through the rest of this lecture.

## 2 How to use exceptions?

### 2.1 Catching an exception

We've already shown you how to catch exceptions in the previous section. Here's the relevant code once again:

```
try {
  int val = readAnIntFromFile_ex(filename);
  /* Assume: lets do stuff with val */
  return doStuff(val);
} catch (const FileNotFoundException& e) {
  // In the end, can't find file
  showAlertMessage("Cannot find a suitable file");
  return;
} catch (...) { // catch all other exceptions
  // try again with next file
  retry;
}
```

Snippet 5: try-catch block

This is known as a try-block. Try-blocks may contain any number of catch clauses. The meaning of a try-block is fairly straightforward: The section enclosed by `try { ... }` is executed under normal (i.e. non-exceptional) circumstances, but some code in there might throw exceptions (in this case, `readAnIntFromFile_ex` might throw `FileNotFoundException`, and `doStuff` might throw exceptions). When an exception is thrown (say from `readAnIntFromFile_ex`), control jumps to the enclosing `catch` clause (i.e. of the nearest try-block), and the exception handler that catches the type of exception that is being thrown.

For example, if a `FileNotFoundException` is thrown, control goes to the exception handler that calls `showAlertMessage`, but if any other type of exception is thrown, control goes to the `catch(...)` clause (i.e. the catch-all handler).

If we did not write the catch-all handler, a thrown exception that is not a `FileNotFoundException` will get thrown out of this try-block to the next enclosing try-block (which can be one or more stack frames away), and will continue to be propagated until a suitable exception handler is found.

### 2.2 Throwing an exception

Let's take a look at the other side — throwing an exception.

To illustrate, we will implement the `read_ex` function in the previous example, which is a wrapper for the `read` system call, but will throw a `ReadException` if the read fails. Let's first implement `ReadException`. In C++, any type can be thrown, so we'll create a simple class that contains a message string.

```
class ReadException {
  std::string m_message;
 public:
  ReadException(std::string message) : m_message(std::move(message)) {}
  const std::string& message() const { return m_message; }
};
```

Snippet 6: `ReadException` implementation

Then we can write `read_ex` like this:

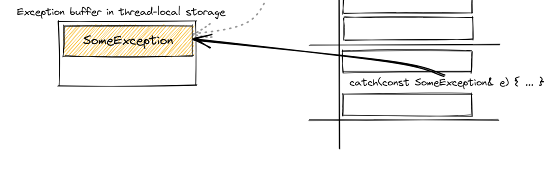
```
ssize_t read_ex(int fd, char* buf, size_t len) {
  ssize_t len_read = read(fd, buf, len);
  if (len_read < 0) {
    throw ReadException("Cannot read: "s + strerror(errno));
  }
  return len_read;
}
```

Snippet 7: `read_ex` implementation

### 2.3 Throw by value, catch by reference

It's a C++ idiom to throw an exception by value, and catch the exception by (usually const) reference, just like in our examples above.

To understand why this idea, we need to understand how the exception object gets passed through the throwing mechanism:



Exceptions are stored in a thread-local buffer

Every thread created in C++ keeps aside a small buffer in thread-local storage. When an exception is thrown, the exception object is copy-initialized into the buffer (using copy construction, move construction, or guaranteed copy elision, depending on the value category of the value being thrown). The stack is then unwound (i.e. traversed) as the exception handling mechanism looks for the nearest suitable exception handler. When the handler is found, the argument of the handler is initialized from the exception in the buffer.

Storing the exception in the buffer is necessary because anything left on the stack will be lost as we unwind the stack, removing stack frames as necessary until we get to an appropriate exception handler. Throwing by value (as opposed to by pointer) ensures that all the information required by the exception is owned by the exception object itself. Throwing specifically by prvalue (i.e. constructing and immediately throwing the exception), like in our `read_ex` example, is very common, and due to guaranteed copy elision, the exception object is constructed in-place inside the buffer. (Note that you can't throw an exception by reference since copy initialization would be performed anyway, so we get a copy of the actual value (rather than a reference) in the buffer.) If you need to throw a pointer, care should be taken to ensure that the pointee is still alive after catching.

Since the exception object is kept in the buffer, catching by reference gives you a reference to that buffer. That is, we do not make any unnecessary copies of the exception object. If we had instead caught the exception by value, then the exception object would be copied into a local variable.

### 2.4 `std::exception` and the C++ exception hierarchy

While you can throw any object, C++ defines `std::exception` and a number of its subclasses for common exception types (see the C++ Reference list in this section's header), and by convention, you should throw an exception that derives (perhaps indirectly) from `std::exception`.

Having inheritance hierarchies are useful because an exception handler for a base class will handle exceptions for any derived class. For example:

```
try {
  if (makeSomeSystemCall() == -1) {
    throw std::system_error(errno, std::system_category());
  }
  /* stuff */
} catch (const std::runtime_error& ex) {
  /* we will catch `std::system_error` here since it is a subclass of `std::runtime_error` */
  return;
} catch (const std::exception& ex) {
  /* catches `std::exception`s that are not `std::runtime_error`s */
  return;
}
```

Snippet 8: try-catch block

A try-block with multiple exception handlers (as in the example above) will cause exception handling to check them in order and pick the first one that matches, and so the handler for `const std::exception&` will only receive non-`std::runtime_error` `std::exception`s.

There two main groups of exceptions that derive from `std::exception` — `std::logic_error` and `std::runtime_error`.

Logic errors are generally things could have been checked explicitly by your code. For example, the `std::stoi` family of functions throw `std::invalid_argument` if the given string is not an integer, and `std::out_of_range` if the integer that the string represents is too large. We could have instead written code to ascertain perhaps that the string contains at most 9 characters, all of which must be between `0` and `9` inclusive, in order to guarantee that the `std::stoi` does not throw:

```
std::string val = /* stuff */
if (val.empty() || val.size() > 9 || std::find_if_not(val.begin(), val.end(), [](
char c) {
  return '0' <= c && c <= '9';
}) == val.end()) {
  std::cout << "Cannot convert to integer" << std::endl;
} else {
  int val = std::stoi(val);
  std::cout << "Converted to integer: " << val << std::endl;
}
```

Snippet 9: try-catch block

Runtime errors, on the other hand, usually represent errors caused by the environment (e.g. the operating system or network). These errors are generally beyond the scope of the program (i.e. your code can't usually fix them).

Note that these are just guidelines — there may be exceptions that don't cleanly fit in one type or the other.

Our `ReadException` class is quite clearly a runtime error (rather than a logic error), since the error is caused by the operating system. We would usually then have `ReadException` extend from `std::runtime_error`:

```
class ReadException : public std::runtime_error {
  std::string m_message;
 public:
  ReadException(const std::string& what_arg) : std::runtime_error(what_arg) {}
  ReadException(const char* what_arg) : std::runtime_error(what_arg) {}
};
```

Snippet 10: `ReadException` inheriting from `std::runtime_error

`what_arg` is a string that is stored in the `std::exception` base class, and an exception handler can obtain that string by calling `ex.what()`.

Another acceptable way (in C++11) would be to have `read_ex` throw `std::system_error`, which is meant to encapsulate a OS-level error code.

### 2.5 Destructors and stack unwinding

Remember how in the very first code snippet (without exceptions), we said that `readAnIntFromFile` will leave the file open if the `read` system call produces an error? Those with a keen eye for detail might also have noticed that in the `readAnIntFromFile_ex` code snippet, our `open_ex` function returns a custom `FileHandle` type instead of a plain `int` (which presumably is an RAII object that closes the handle when it is destructed). These hint at something else that happens during stack unwinding. When an exception is thrown, stack frames are popped off the stack until the exception handler is reached. But every time a stack frame is removed, local variables in that stack frame are destructed (by calling their destructors, if any).

For example, if our `FileHandle` looked like this:

```
class FileHandle {
  int fd;
 public:
  FileHandle(int fd) : fd(fd) {}
  ~FileHandle() {
    close(fd);
  }
}
```

Snippet 11: Possible implementation of 'FileHandle'

Then the destructor, which closes the file, will be called whether control leaves the function normally or via an exception.

(Note: In practice, we might instead write a class `File` that also encapsulates operations on the file, such as read and writing the contents of the file.)

Why is this behaviour "acceptable"? As we have talked about in previous lectures, destructors are usually used to enforce ownership of resources. These resources then are no longer useful once the object is destroyed, whether or not it was able to accomplish the task successfully. This kind of behaviour is also used in programming languages without RAII. In such languages, the language usually specifies a `finally` clause that is executed whether control leaves normally or via an exception, and this is where resources are usually released.

▼ **Throwing out of a destructor**

Destructors are unique pieces of code that can be called while the stack is being unwound. This raises the question about thrown an exception from a destructor. What happens if the destructor code throws an exception that goes out of the destructor?

In C++, you can't have two exceptions being thrown at once. The standard instead says that if the destructor throws an exception during stack unwinding, `std::terminate` will be called. `std::terminate` calls `std::abort` (which terminates the program abnormally) by default, though it is possible to install a handler, usually to perform cleanup operations (it is generally impossible to gracefully recover from `std::terminate`, as it usually indicates a logic bug).

In practice, you should almost never throw an exception from the destructor, since the destructor might have been called due to another exception. The standard also encourages this by making destructors `noexcept` (i.e. they can't throw exceptions) unless any of their bases classes or member fields have a destructor that is not `noexcept`.

### 2.6 `noexcept`

The `noexcept` keyword has two purposes — it is both a specifier and an operator.

#### 2.6.1 The `noexcept` specifier

Writing `noexcept` after a function declaration says that exceptions will not be thrown out of the function. More precisely, if any exception from within the function attempts to unwind the stack past this function's stack frame, then unwinding will stop and `std::terminate` will be called instead.

`noexcept` is part of the function type since C++17, just like `const` for member functions.

It is written like this:

```
int doStuff(int x) noexcept {
  /* stuff */
}
```

Snippet 12: Writing a noexcept function

Member functions can also be noexcept, and where it is also const, the `const` comes before the `noexcept`:

```
struct Point {
  double hypot() const noexcept {
    /* stuff */
  }
};
```

Snippet 13: Writing a const noexcept member function

Note that this does not mean that no exceptions can be thrown within the function — it is perfectly legal to throw exceptions within a function declared `noexcept`, as long as all exceptions are handled so they don't leave the function. In other words, `noexcept` describes the function call boundary, and not the contents of the function.

Why do we want to do this? Making a function `noexcept` serves two purposes:

1. It makes it easier for the programmer to reason about code, since they can see if certain functions are guaranteed to never throw exceptions.
2. It aids optimisations, since the compiler does not need to generate code to handle an exception being thrown out of that function.

The main drawback is the verbosity of writing `noexcept` on every such function call.

In practice, programmers tend to pay more attention to adding `noexcept` for general-purpose library code used by many parts of the program, but may omit to write them for more high-level functions that are not general-purpose.

#### 2.6.2 `noexcept(bool)`

It's possible to make the noexcept-ness of a function depend on any compile-time expression, by writing `noexcept(expr)` where `expr` is any expression that can be evaluated to a boolean at compilation time. `noexcept(true)` is equivalent to `noexcept`, while `noexcept(false)` is equivalent to not writing any noexcept specifier at all (i.e. the function may throw exceptions).

It's possible to pass any constant expression, but often we want the noexcept-ness of the function to depend on the noexcept-ness of certain expressions within the function. For example, the standard provides a type trait `std::is_nothrow_copy_constructible_v<T>` which evaluates to `true` if `T`'s copy constructor is noexcept, and `false` otherwise. We'll cover the details of `std::is_nothrow_copy_constructible_v<T>` and other type traits in a separate lecture, but for now just know that `std::is_nothrow_copy_constructible_v<T>` evaluates to a compile-time constant, either `true` or `false`, depending on the type `T`. We would then write `noexcept(std::is_nothrow_copy_constructible_v<T>)`, for example:

```
template <typename T>
int doStuff(T x) noexcept(std::is_nothrow_copy_constructible_v<T>) {
  /* stuff */
  // perhaps we want to do copy construction of T here
  T y = x;
}
```

Snippet 14: Writing a noexcept function

#### 2.6.3 The `noexcept` operator

In the previous example, we made the noexcept-ness of a function depend on whether `T`'s copy constructor is noexcept. But what if we wanted it to depend on something more complicated? Perhaps we wrote `a.val() == b.val()` in the function, and this expression may or may not be noexcept. How would we check that every part of this expression is noexcept?

C++ provides a convenient way to do this — the `noexcept` *operator*. The `noexcept` operator takes in an expression, and produces `true` if that expression is noexcept, and `false` otherwise. This is a compile-time operation — the given expression is never evaluated, and the value produced by the noexcept operator depends on whether every operation in the expression is specified as `noexcept`. Note that this is not about whether an exception is actually thrown; instead it is about whether all parts of the function are declared as noexcept or not.

We could write something like this (not that the expression `a.val() == b.val()` is never actually evaluated):

```
bool is_noexcept = noexcept(a.val() == b.val());
```

Snippet 15: `noexcept` operator

Since this is a compile-time operation, its value is known at compilation time, and so we can use it anywhere a compile-time constant is expected:

```
// noexcept expression in a non-type template parameter
std::array<int, noexcept(a.val() == b.val()) ? 10 : 20> arr;

// noexcept expression assigned to constexpr variable (a variable whose value is
 known at compile time)
constexpr bool cond = noexcept(a.val() == b.val());

// A constexpr variable can later be used anywhere a compile-time constant is exp
ected
std::array<int, 15 + cond> arr2;
// no need to capture a constexpr variable
// (unless you have a buggy compiler: https://stackoverflow.com/questions/4261042
9/must-constexpr-expressions-be-captured-by-a-lambda-in-c)
std::partition(v.begin(), v.end(), [](int x) { return x % 2 == (cond ? 0 : 1);
});
constexpr int new_val = cond ? 1 : 2;
```

Snippet 16: `noexcept` operator in places where a compile-time constant is expected

#### 2.6.4 `noexcept(noexcept(...))`

You might have guessed that the main use case of the `noexcept` operator is to determine if the function performing that expression should be marked as noexcept. In other words, you might want to write a function like this:

```
void f(const Stuff& a, const Stuff& b) noexcept(noexcept(a.val() == b.val())) {
  /* do stuff that includes the expression `a.val() == b.val()`
}
```

Snippet 17: `noexcept(noexcept(...))` example

It looks repetitive, but the two `noexcept`s have different meanings — the first `noexcept` is a specifier, while the second `noexcept` is an operator.

# 3 Exception safety

Exception safety, or exception guarantee, is a property of a function. It tells us what we can assume about the state of the objects potentially modified by the function. We usually see this when taking about member functions of a class.

Consider the `NullableOwnPtr` example from Lecture 6. We want to implement `operator=(const T&)`, so that we can do assignment from `T`, similar to `std::optional`. In other words, we want something like this to work, assuming that `MyClass` is copy constructible:

```
struct MyClass { /* class definition */ };

NullableOwnPtr<MyClass> nop = /* expression */;
MyClass new_stuff = /* expression */;

// We want this to work and set `nop` to contain a copy of `new_stuff`:
nop = new_stuff;
```

Snippet 18: Use case of assignment from the element type

Here's one way to implement this `operator=`:

```
template <typename T>
struct NullableOwnPtr {
private:
  T* m_ptr;

public:
  /* member functions go here */
};
```

```
// The assignment operator from `T`
NullableOwnPtr& operator=(const T& val) {
  auto copy = NullableOwnPtr{val};
  this->swap(copy);
  return *this;
}
```

Snippet 19: Implementing `operator=(const T&)` in the easiest way

But is this the best we can do?

It isn't, because we're making a new heap allocation and then deleting the old one. If the `NullableOwnPtr` already contains some value, then we could have just copied the new value into the existing allocation, without making any additional allocations. Something like this:

```
// The assignment operator from `T`
NullableOwnPtr& operator=(const T& val) {
  if (this->m_ptr) {
    // use the old heap allocation
    this->m_ptr->~T(); // destroy the object
    new (this->m_ptr) T{val}; // copy construct the new object into the existing
 heap allocation
  } else {
    // create a new heap allocation
    this->m_ptr = new T{val};
  }
  return *this;
}
```

Snippet 20: More efficient way of implementing `operator=(const T&)`

And we now have more efficient code.

This is all good if nothing here throws exceptions, because the entire function is guaranteed to be executed till completion.

The problem however comes because we don't know what `T` is. Since we don't know anything about `T`, we can't be sure that `T`'s member functions don't throw exceptions. In this case, we're concerned about the copy constructor of `T`, which is invoked in the line `new (this->m_ptr) T{val}` to copy the object. What would happen if `T`'s copy constructor threw an exception there?

Well, it would be really bad. This is because we've destructed the old object but didn't put a new object in its place, so there wouldn't actually be a valid object in the heap allocation. This invokes undefined behaviour and would likely lead to a crash when we subsequently attempt to use or destruct the invalid object.

In this example, if `T` had a copy assigment operator, we could have written this instead:

```
// The assignment operator from `T`
NullableOwnPtr& operator=(const T& val) {
  if (this->m_ptr) {
    // use the old heap allocation
    *(this->m_ptr) = val; // copy assign the new object into the existing heap al
location
  } else {
    // create a new heap allocation
    this->m_ptr = new T{val};
  }
  return *this;
}
```

Snippet 21: More efficient way of implementing `operator=(const T&)`, using copy assignment of `T`

The exception safety of this code would then depend on what happens when the copy assignment of `T` throws an exception (or in other words, the exception safety of copy assigning `T`) — does it leave the assignee in its old state? Or does it leave it in an arbitrary state?

## 3.1 The levels of exception safety

So now with the example above, we have adequate motivation to formalise the levels of exception safety of a function (or informally, what happens to the objects being modified when an exception is thrown).

There are four levels of exception safety, listed from the strongest guarantee to the weakest guarantee:

1. Nothrow exception guarantee — the function never throws exceptions. (This can be indicated in code by writing `noexcept`.)
2. Strong exception guarantee — if the function throws an exception, then nothing is modified by the function. (If something has already been modified, it must be rolled back to its original state.)
3. Basic exception guarantee — if the function throws an exception, the program remains in a valid state, i.e. no resources are leaked and all objects are in a valid state.
4. No exception guarantee — no guarantees about anything after an exception is thrown.

Functions that do not throw are the easiest to use, since we do not need to consider any additional code paths. If possible, functions should not throw exceptions.

If we write a function that can throw exceptions, we should really only consider making it satisfy either the strong or basic exception guarantee. This is because having no exception guarantee means that it's impossible to properly recover from an exception. If a function has no exception guarantee, we can't use any of the modified objects anymore (this also means we can't even destroy them, because the destructor is allowed to assume that the object is in a valid state). Between the strong and basic exception guarantees, the strong exception guarantee is preferred, but in some cases it is difficult or impossible to give your function the strong exception guarantee, in which case we settle for the basic exception guarantee.

### 3.1.1 Adding the strong exception guarantee to `push_back` in `SimpleVector`

Remember that saying that an object is in a *valid state* is saying that the invariants of an object are satisfied, which means that you can still call functions (with no preconditions) on it. For example, in our `SimpleVector` example (member fields reproduced below), a valid `SimpleVector` is one where `m_buffer` points to an array of exactly `m_size` `ElemTy` elements.

```
struct SimpleVector {
private:
  ElemTy* m_buffer;
  size_t m_size;
};
```

Snippet 22: `SimpleVector` member fields

So if we say that a function operating on a `SimpleVector` satisfies the basic exception guarantee, we mean that even if the function throws an exception, `m_buffer` must still point to an array of exactly `m_size` `ElemTy` elements, AND no resources are leaked.

Recall this `push_back` member function in `SimpleVector` (slightly modified to get rid of the moves for now):

```
void push_back(const ElemTy& element) {
  ElemTy* new_buffer = new ElemTy[m_size + 1];
  for (size_t i = 0; i < m_size; i++) {
    new_buffer[i] = m_buffer[i];
  }
  new_buffer[m_size] = element;
  m_size += 1;

  delete[] m_buffer;
  m_buffer = new_buffer;
}
```

Snippet 23: `push_back` member function in `SimpleVector` from a previous lecture

Does this function satisfy the basic exception guarantee?

Let's analyse the code and pick out the operations that might throw exceptions:

```
void push_back(const ElemTy& element) {
  ElemTy* new_buffer = new ElemTy[m_size + 1]; // <-- value-initializing `ElemTy`
calls the default constructor if it exists, which might throw
  for (size_t i = 0; i < m_size; i++) {
    new_buffer[i] = m_buffer[i]; // <-- copy assignment of `ElemTy` might throw
  }
  new_buffer[m_size] = element; // <-- copy assignment of `ElemTy`
  m_size += 1;

  delete[] m_buffer;
  m_buffer = new_buffer;
}
```

Snippet 24: `push_back` member function in `SimpleVector` from a previous lecture, annotated with potential exception sites

If any of those places throw an exception, will the vector remain in a valid state? Notice that `m_size` and `m_buffer` (as well as the data pointed to by `m_buffer`) will only be modified after we've passed all the code that might throw exceptions. This means that if an exception is thrown, the vector is guaranteed to be unmodified.

However, this code will leak memory if an exception is thrown from the copy assignment of `ElemTy`, because `new_buffer` won't be freed, so this code doesn't even satisfy the basic exception guarantee.

An easy way to remedy this problem is to add a try-block:

```
void push_back(const ElemTy& element) {
  ElemTy* new_buffer = new ElemTy[m_size + 1]; // <-- if default construction thr
ows, the buffer will be freed automatically
  try {
    for (size_t i = 0; i < m_size; i++) {
      new_buffer[i] = m_buffer[i];
    }
    new_buffer[m_size] = element;
  } catch (...) {
    delete[] new_buffer;
    throw; // <-- re-throws the original exception
  }
  m_size += 1;

  delete[] m_buffer;
  m_buffer = new_buffer;
}
```

Snippet 25: `push_back` with strong exception guarantee using a try-block

Since C++ has RAII instead of `finally` clauses, we can rewrite this in a more idiomatic way using `std::unique_ptr`:

```
void push_back(const ElemTy& element) {
  std::unique_ptr<ElemTy[]> new_buffer(new ElemTy[m_size + 1]); // note: for more
idiomatic code, use `std::make_unique<ElemTy[]>(m_size + 1)` instead
  for (size_t i = 0; i < m_size; i++) {
    new_buffer[i] = m_buffer[i];
  }
  new_buffer[m_size] = element;
  m_size += 1;

  delete[] m_buffer;
  m_buffer = new_buffer.release(); // <-- takes the raw pointer out of the `std::
unique_ptr`
}
```

Snippet 26: `push_back` with strong exception guarantee using `std::unique_ptr`

And now our `push_back` member function satisfies the strong exception guarantee.

### 3.1.2 `std::move` and the strong exception guarantee

Why did we use a version of `push_back` that performs copy assignment instead of move assignment? This is because move assignment is more nuanced.

Move assignment modifies the object being moved from, which means that by the time an exception is thrown, the contents of the old buffer might have been modified. This means that if we wrote `push_back` like this…

```
void push_back(const ElemTy& element) {
  std::unique_ptr<ElemTy[]> new_buffer(new ElemTy[m_size + 1]);
  for (size_t i = 0; i < m_size; i++) {
    new_buffer[i] = std::move(m_buffer[i]); // <-- move assignment (modifies `m_b
uffer[i]`)
  }
  new_buffer[m_size] = element; // <-- copy assignment
  m_size += 1;

  delete[] m_buffer;
  m_buffer = new_buffer.release();
}
```

Snippet 27: Optimised `push_back` with basic exception guarantee

… it would only satisfy the basic exception guarantee (but not the strong exception guarantee).

What could we do to make it satisfy the strong exception guarantee? The main problem here is if move assignment or copy assignment throws, then the earlier iterations of the loop might have already modified the elements in the old buffer.

Class authors may want copy assignment or copy construction to throw, since they may need to acquire resources (e.g. heap memory, like `std::string`, or locks, like `std::lock_guard`). However, should move assignment throw? Or perhaps, are there reasonable situations where you might want move assignment or move construction to throw? Since moving an object "steals" the resources of the moved-from object, there should not be a need to acquire new resources. (I have not come across a situation where you would actually want move construction or move assignment to throw, and if you can think of something, I'd be happy to talk about it after the lecture.)

So to simplify things, let's assume that moves never throw. Is there now a better way to do things so that `push_back` satisfies the strong exception guarantee?

We could do the copy assignment before all the move assignments, so that by the time we get to the moves, we know that the operation must succeed:

```
void push_back(const ElemTy& element) {
  std::unique_ptr<ElemTy[]> new_buffer(new ElemTy[m_size + 1]);
  new_buffer[m_size] = element; // <-- copy assignment
  for (size_t i = 0; i < m_size; i++) {
    new_buffer[i] = std::move(m_buffer[i]); // <-- move assignment (modifies `m_b
uffer[i]`)
  }
  m_size += 1;

  delete[] m_buffer;
  m_buffer = new_buffer.release();
}
```

Snippet 28: Optimised `push_back` with strong exception guarantee, assuming moves don't throw

**▼ On nofail moves and trivial relocatability**

In C++, move constructors and move assignment operators are normal functions that may be customised in any way, including throwing an exception. However, throwing an exception is unexpected in almost all situations, and most moves simply perform a memberwise move and reset the state of the moved-from object to something similar to the default-constructed state (if necessary). Furthermore, the moved-from object is most of the time destructed immediately after its state is moved from it.

This makes some other programming languages (notably Rust) settle on simpler (though perhaps more rigid) move semantics. In those languages, moving an object is equivalent to a bitwise copy (i.e. `memcpy`), and the original object no longer exists after a move (i.e. the destructor should not be called). While more rigid, most classes that want to be movable can be written to satisfy such semantics, and those classes that can't satisfy such semantics (e.g. due to internal references) can implement some arbitrary member function to perform the atypical move. Memcpy moves, in this case, are more "pure" in the sense that it feels like there's really only one object in existence, and we're just moving it around in memory. (C++ move semantics are more like creating a new object and stealing the resources of the old one.)

The ease of reasoning about memcpy moves has led some to propose a "trivial relocatability" trait in C++, where classes can opt in to this trait to declare that calling the move constructor followed by destructing the moved-from object is equivalent to a bitwise copy. Library implementors can then optimise containers or wrappers of trivially relocatable types to actually use `memcpy` to move them, even if these types are not trivially copyable.

#### 3.1.2.1 `std::move_if_noexcept` and `std::vector::push_back`' exception guarantees

`std::vector::push_back` guarantees the strong exception guarantee whether or not moves can throw exceptions. This means that if moves might throw exceptions (i.e. the move constructor and move assignment operator are not `noexcept`), then `push_back` copies instead of moves the existing elements to the new buffer.

Because of this, as you design a class, you should tag move constructors and move assignment operators with `noexcept` whenever possible, so as to get performance benefits when instances of your class are placed in containers like `std::vector`.

The C++ standard library helpfully provides `std::move_if_noexcept` — it is equivalent to `std::move` if move construction is `noexcept`, and equivalent to a copy otherwise. (In other words, `std::move_if_noexcept` converts the given reference to an rvalue reference if move construction is `noexcept`, and a const lvalue reference otherwise.) This utility function can then be used in container classes such as `std::vector` to always provide the strong exception guarantee but perform optimisations when possible.

#### 3.1.2.2 Standard library exception guarantees

In general, member functions of containers of the standard library satisfy the strong exception guarantee when adding an element, and satisfy the nofail exception guarantee when removing an element, unless it is impossible to perform.

Some situations where it is impossible to make the strong exception guarantee are: - Inserting to the middle of a `std::vector` or `std::deque` using `std::insert` or `std::emplace` - In-place construction of a value into a `std::optional` using `std::emplace`

These situations generally arise because of the need to perform some modifying operation on the original data structure before the new element can be constructed in its desired location.

Furthermore, moves and swaps of standard library containers are `noexcept` (apart from `std::array`), which allows them to be used efficiently in other standard library containers.

# 4 How are exceptions implemented?

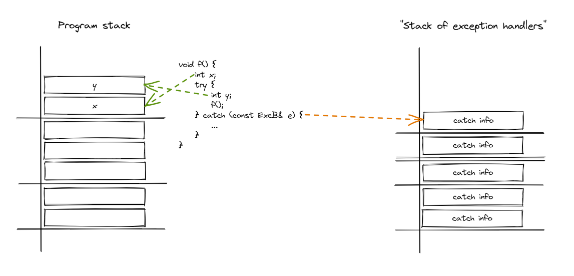
Thus far, exceptions have been running on magic — we've simply assumed stack unwinding works, and it is possible to traverse the stack, calling destructors and exiting stack frames on the way, and find a matching exception handler. If you think stack unwinding is "trivial to implement", think again.

How does the stack unwinding mechanism know which destructors to run and which catch block can handle the exception? What happens if the exception is being thrown out of several layers of function calls? It isn't immediately clear how to traverse the stack, since the objects on the stack do not have any type or function information encoded with it.

## 4.1 A simple exception handling mechanism

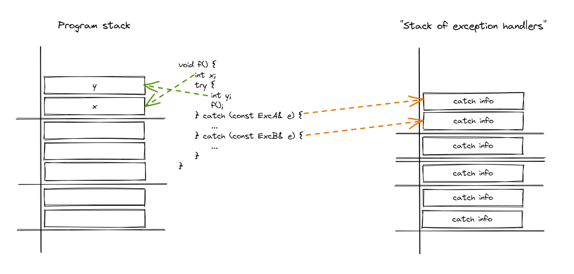
We'll start by describing a "simple" way to implement exception handling. It is a hypothetical exception handling mechanism that isn't used on any platform as far as I know.

Notice that just like the call stack, we have a conceptual stack of exception handlers (i.e. the catch clause of a try block), each nested inside the previous one. This stack of exception handlers spans the entire call hierarchy of the program, and there may be multiple exception handlers within the same function. When control enters a try block, an exception handler is pushed onto this stack, and when control leaves, the exception handler is removed. For example:



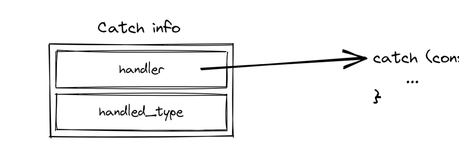

Conceptual stack of exception handlers

Since a try-block with multiple catch clauses can be rewritten as nested try-blocks without changing the semantics of the program, we can extend this concept to describe catch clauses with multiple try-blocks:



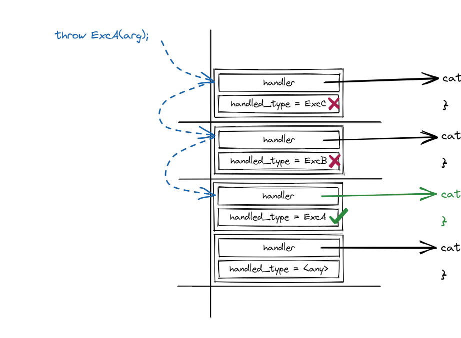

Conceptual stack of exception handlers, extended

Each "catch info" structure would contain a pointer to the exception handler code (think of this as a function synthesised by the compiler), as well as type information describing the type of exception this handler accepts:




"Catch info" struct

When an exception is thrown, this stack of exception handlers is traversed to arrive at the nearest matching exception handler:




Finding the correct exception handler

But there are two problems here:

- The program stack hasn't been restored to what the exception handler expects — specifically, the base pointer and stack pointer (`rbp` and `rsp` on x64) has to be restored to the stack frame in which the catch clause is located.
- The destructors of objects that should no longer exist at the point of the exception handler have not yet been called.

The second problem can be fixed easily. Notice that during stack unwinding, calling the destructor of an object is equivalent to a catch-all handler that always rethrows the exception:

```
catch (...) {
  obj.~Obj();
  throw;
}
```

Snippet 29: Synthesised "exception handler" performing destruction of an object

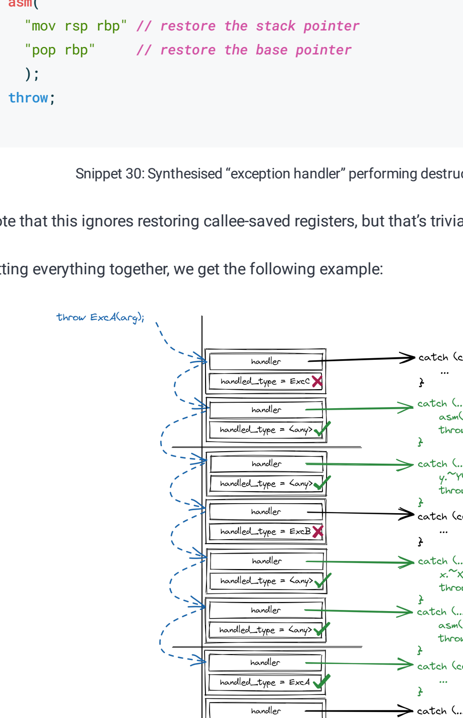
Restoring the base pointer and stack pointer is slightly more complex, but it can also be done by installing a catch-all handler that rethrows (as part of the prologue of each function call):

```
catch (...) {
  asm(
    "mov rsp rbp" // restore the stack pointer
    "pop rbp"     // restore the base pointer
    );
  throw;
}
```

Snippet 30: Synthesised "exception handler" performing destruction of an object

(Note that this ignores restoring callee-saved registers, but that's trivial to add to the handler.)

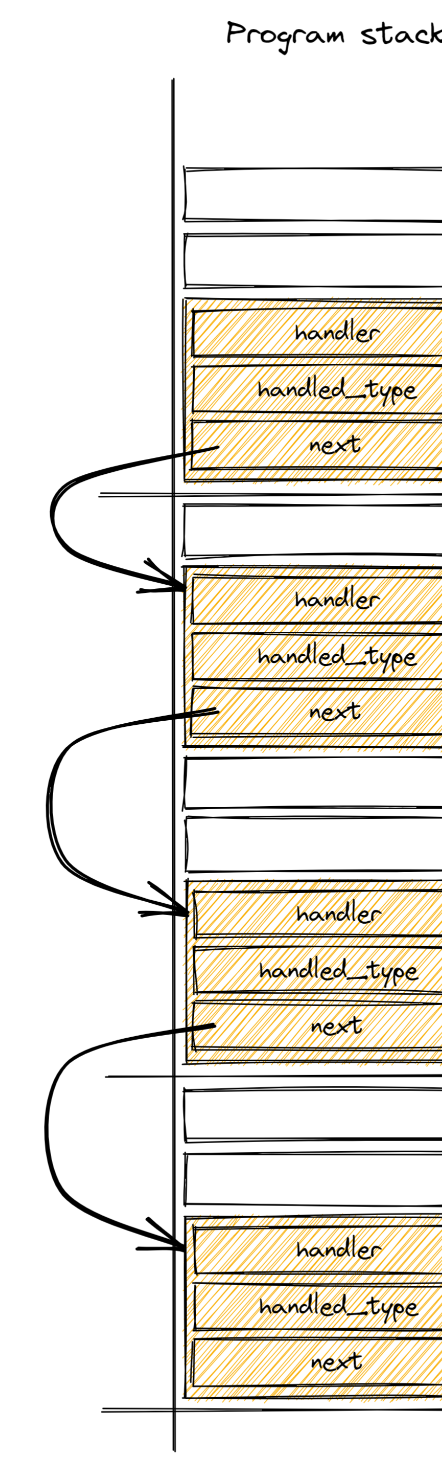
Putting everything together, we get the following example:




Finding the correct exception handler, with destructors and returns

At this point, we have a workable exception handling mechanism. However, having two separate stacks is rather wasteful, and we would rather combine them into just one. (This will be further motivated later where we make some optimisations to push less blocks onto the exception handling stack.)

It's possible to combine the two stacks by converting the exception handling stack into a linked list embedded in the program stack:




Embedding the exception handling stack into the program stack

If you think about this exception handling mechanism carefully, you might realise that we've glossed over what the "handled type" stores, and how it could possibly work with the exception hierarchy.

If we don't need an inheritance hierarchy for our exceptions, then we can just assign each exception type a unique identifier (chosen at compilation time), such as the one chosen by `std::type_info`. Then the "handled type" field in the catch info stores this identifier, which can be checked for equality. If we only had single inheritance, we could generate a table at compile time that maps each type identifier to its parent. Multiple inheritance is somewhat more complicated, and it will be covered in Lecture 10.

And we now have a decent exception handling mechanism.

The next two subsections describe how to augment this sections to improve the efficiency of the exception handling mechanism.

## 4.2 Per-function handler (personality routine)

The first thing one should notice is that these *catch info* structs take up a sizeable amount of space on the stack. This is because you need one of them for every non-trivial destructor and every catch block you have.

We want to reduce the amount of space we need, and the amount of operations we need to do in the non-exceptional case. Within the same stack frame, can we somehow collapse the structs into just a single one?

It turns out we can! With some tracking of which part of the function is currently being executed (using just a single integer), it's possible to figure out which exception handlers need to be run.

Let's take a look a this function, for example:

```
std::vector<int> f(const Database& db) {
  std::vector<int> result;
  for (size_t i = 0; i != 10; ++i) {
    try {
      // db.query(i) might throw IndexNotFoundException or ConnectionException (w
hich will be thrown to the caller)
      int val = db.query(i);
      result.push_back(val);
    } catch (const IndexNotFoundException& ex) {
      result.push_back(-1);
    }
  }
  return res;
}
```

Snippet 31: Example code for investigating exception handling

Going by the simple exception handling scheme described earlier, we would push and pop *catch info* structs at the following locations:

```
std::vector<int> f(const Database& db) { // <-- push catch info to return from fu
nction
  std::vector<int> result;
  // <-- push catch info to destruct `result`
  for (size_t i = 0; i != 10; ++i) {
    try { // <-- push catch info to catch `IndexNotFoundException`
      int val = db.query(i);
      result.push_back(val);
    } catch (const IndexNotFoundException& ex) {
      result.push_back(-1);
    } // <-- pop catch info to catch `IndexNotFoundException`
  }
  return res;
  // ^-- performed while returning:
  //     - pop catch info to destruct `result`
  //     - pop catch info to return from function
}
```

Snippet 32: Example code with annotations for simple exception handling

Notice that this creates code regions where each code region is associated with a certain state of the stack of exception handlers (highlighted using different colours):

```
std::vector<int> f(const Database& db) { // <-- push catch info to return from fu
nction
  std::vector<int> result;
  // <-- push catch info to destruct `result`
  for (size_t i = 0; i != 10; ++i) {
    try { // <-- push catch info to catch `IndexNotFoundException`
      int val = db.query(i);
      result.push_back(val);
    } catch (const IndexNotFoundException& ex) {
      result.push_back(-1);
    } // <-- pop catch info to catch `IndexNotFoundException`
  }
  return res;
  // ^-- performed while returning:
  //     - pop catch info to destruct `result`
  //     - pop catch info to return from function
}
```

Snippet 33: Example code with code ranges highlighted

Observe that the code regions are regions over the *code* of the function, which is something that can be resolved statically at compilation time. It does not matter that we're doing some kind of loop here, or any form of runtime control flow. This is because the state of the exception handler stack depends only on which region of code threw the exception.

We hence need only to determine which code region the exception was thrown from. To do so, we first associate each code region with a fixed integer. Then, we synthesise a new `int` variable, and at every point where control flow can enter the code region, we set this variable to the associated integer of the code region. This is the same code as above, but with the synthesised variable indicating the current code region:

```
std::vector<int> f(const Database& db) { // <-- push catch info to return from fu
nction
  int region = 0; // <--
  std::vector<int> result;
  region = 1; // <--
  for (size_t i = 0; i != 10; ++i) {
    try {
      region = 2; // <--
      int val = db.query(i);
      result.push_back(val);
      region = 1; // <--
    } catch (const IndexNotFoundException& ex) {
      result.push_back(-1);
    }
  }
  return res;
}
```

Snippet 34: Example code with code region tracker

Now, whenever an exception is thrown, we read the value of the `region` variable, and use some kind of lookup table to determine the list of handlers to check. Exactly how this lookup table is implemented varies, but this lookup table is always determinable during compilation, and hence can be compiled into the executable (usually in a read-only static memory region of the program). The code that checks the lookup table and (based on the value of `region`) invokes the exception handlers is known as the *personality routine* of this function — each function that performs operations that might throw exceptions will have a personality routine.

Compared to the the previous section, this per-function handling mechanism is a fairly large performance boost. From pushing a struct onto the stack every time control flow enters a new code region, we've simplified it to merely updating an integer (which is likely to reside in a register). Note that when we talk about performance here, it's performance in the non-exceptional case (i.e. when no exceptions are thrown). In the exceptional case using a per-function handler is likely to be slower than our simple exception handling mechanism, because of the need to use the lookup table to find the correct exception handlers to invoke. However, since exceptions are assumed to occur only in "exceptional" situations, the performance of the exceptional case is taken to be a non-issue. Hence, we are willing to make a trade-off and improve performance in the non-exceptional case at the cost of poorer performance in the exceptional case.

## 4.3 Zero-cost exception handling

Even though we've reduced the non-exceptional cost of exceptions to updating a single integer whenever we need to push or pop an exception handler, this is still a somewhat significant amount of work in the non-exceptional code path. Zero-cost exception handling takes this a step further — in the non-exceptional code path, we do not want to execute any additional code *at all*!

How is that even possible?

As evidenced by the use of the `region` variable from the previous section, we only need to find *some* way to figure out the current code region when an exception is thrown. The exact way doesn't matter, but we want to spend as little effort as possible in the non-exceptional code path.

What else can be used to find which code region we are currently in? The instruction pointer! Since the code regions are simply ranges of instructions, having the current instruction pointer is sufficient (given an appropriate lookup table) to determine the code region we are currently in, and thus the list of exception handlers we need to check. The personality routine would then take in the current instruction pointer (instead of the `region` variable), and the lookup table will then map instruction ranges to something equivalent to the "catch info" struct we saw earlier.

### 4.3.1 Throwing out of a function

While unwinding the stack, it is possible to unwind out of a function (i.e. throw an exception out of a function). It isn't possible to directly specify in the lookup table which handlers to use in the parent function, since there may be many functions that could possibly call the current function.

However, since the return address of a function invocation is saved on the stack (so that we can return to the caller in non-exceptional execution), we can use the same return address to figure out the caller function, and hence obtain its personality routine. We can then invoke that personality routine, with the return address as the current instruction pointer.

# 5 Alternatives to exceptions

Exceptions aren't the only way to handle errors. C APIs return an error code to indicate a failure condition. There are a few other error handling mechanisms that have been explored by programming language designers in detail, which may be used in place of exceptions, depending on the kind of error we want to handle or the kind of software we are writing.

This section will briefly discuss two alternatives to exceptions that are being explored in the C++ world — `std::expected` and contracts.

## 5.1 `std::expected` (C++23)

`std::expected<T, E>` is a discriminated union (i.e. a union together with a flag that indicate which alternative it holds, like a `std::variant`) of types `T` and `E`. `T` is the "success" type and `E` is the "failure" type. `std::expected<T, E>` is meant to be the return type of a function that might fail. For example:

```
std::expected<size_t, std::string> read_ex(int fd, char* buf, size_t bufsz) {
  size_t res = read(fd, buf, bufsz);
  if (res < 0) {
    // Return the failure alternative
    return std::unexpected(strerror(errno));
  } else {
    // Return the success alternative (implicit conversion)
    return res;
  }
}
std::expected<int, std::string> readAnIntFromFile_ex(int fd) {
  char buf[64];
  if (auto res = read_ex(fd, buf, 64); !res) {
    return std::unexpected(res.error());
  }
  /* parse the integer from the buffer */
}
```

Snippet 35: `read_ex` with `std::expected`

There are two main benefits (or drawbacks, depending on how you see it) between `std::expected` and exceptions:

Firstly, `std::expected` forces the programmer to check for failure at each call site (i.e. the if-statement that checks `!res`). This reduces the likelihood of forgetting to handle some exception (which would otherwise result in the program aborting), which results in safer code. However, this is a significant increase in verbosity at the call site as compared to using exceptions. The verbosity however is mostly due to the C++ implementation of `std::expected` — this a programming language concept isn't inherently verbose, as Rust's `std::result::Result<T, E>` implements this concept in a much less verbose manner:

```
fn read_ex(fd: i32, buf: &mut [u8]) -> Result<usize, String> {
    let res: isize = read(fd, buf);
    if res < 0 {
        Err(strerror(errno))
    } else {
        Ok(res as usize)
    }
}
fn readAnIntFromFile_ex(fd: i32) -> Result<i32, String> {
    let mut buf: [u8; 64];
    let len: usize = read_ex(fd, &mut buf)?; // the '?' early-returns if there is an error
    /* parse the integer from the buffer */
}
```

Snippet 36: Rust `Result<T, E>` example

Secondly, for the cost of slightly larger return values and a branch at each call site, the overhead of handling an error is essentially eliminated. This means that we may use `std::expected` for commonly-occuring error conditions, not just "exceptional" ones. However, slightly larger return values may mean that the return value is passed in memory instead of a register, which may incur a noticeable slowdown. Sutter's paper (see "Further reading" section) suggests using an unused register for the discriminant (i.e. the flag), so that the size of the returned object remains the same as with traditional exceptions, but it is not clear if any of the major compiler vendors will implement this.

## 5.2 Contracts (planned for a future C++ standard)

Contracts, or *design by contract*, is a programming paradigm first seen in the Eiffel programming language. This section will focus on how contracts can be a replacement for certain kinds of exceptions.

As we have discussed earlier, `std::logic_error` is the standard-provided base class for error conditions that could have been checked by the caller before making the function call. We gave the example of `std::stoi`, which throws something that inherits from `std::logic_error` if the given string is not interpretable as an integer.

However, who is responsible (the caller or callee) for the error when the string is not interpretable as an integer? By having the callee (`std::stoi`) check for this error condition and throw an exception when in happens, we're putting the responsibility on the callee. But perhaps it's the caller's responsibility for the mess that it passed to the callee? The key idea of contracts is to pin the blame squarely on the caller for handing the callee a bad input, and to provide the language support necessary for formalising what the callee demands of the caller (this is the *contract* between the caller and the callee). You may have heard of *preconditions*, which typically written in comments on the function declaration. Contracts, as a language feature, allows the programmer to write these preconditions in code, and have the compiler (optionally) assert them.

For example, we could create a version of `std::stoi` that demands that all characters in the string are between `'0'` and `'9'`:

```
int my_stoi(const std::string& s)
[[ pre: std::all_of(s.begin(), s.end(), [](char c) { return '0' <= c && c <= '9';
}) ]]
{
  int x = 0;
  for (char c : s) {
      x *= 10;
      x += c - '0';
  }
  return x;
}
```

Snippet 37: Contracts example

A good contracts implementation should also provide support for *postconditions*, which are the guarantees that the callee makes about its return value and the final state of its arguments (for arguments that allow modification to objects not owned by the callee, such as pointers and references).

Depending on the compilation mode, the preconditions and postconditions behave in one of these three ways: - Assertion: Code will be emitted to check preconditions when entering the function and postconditions when leaving the function. This is useful when debugging code. - Ignored: The preconditions and postconditions are ignored. - Assumption: The compiler will assume that the preconditions and postconditions are true, in the sense that it is allowed to perform optimisations that assume those conditions (i.e. it is undefined behaviour if preconditions or postconditions are not satisfied).

Contracts was originally slated for C++20, but now has been delayed indefinitely, supposedly due to disagreements about the three ways (above) one can compile the preconditions and postconditions.

## 5.3 Further reading

There is a great paper on zero-overhead deterministic exceptions by Herb Sutter that discusses different error handling mechanisms and their benefits and drawbacks.

# 6 References

- Exception Handling in LLVM
- How a C++ compiler implements exception handling (Per-function handler (not zero-cost))
- Exceptions under the hood (Zero-cost exception handling)