Exceptions

* Based on notes by Brad Lushman, used with ... I'll ask later.

Suppose `v` is a `vector<int>`.

- `v[i]` is the `i`-th element of `v`. Out-of-bounds is unchecked. If `i` goes out-of-bounds, undefined behaviour.
- `v.at(i)` is the `i`-th element of `v` but out-of-bounds is checked.

What happens if `v.at(i)` goes out of bounds?

- `vector`'s code can detect the error but doesn't know what to do.
- client is the one who should decide what to do, how to recover, etc but can't detect the error.

$\Rightarrow$ Error recovery is inherently a non-local problem.

When an error condition arises in C++, the function ***raises an exception***.

Exceptions

Default behaviour: Execution stops.

However, we can write ***handlers*** to ***catch*** exceptions and deal with them.

`vector<T>::at` throws an exception of type `std::out_of_range` when it fails. We can write a handler to catch this exception.

```
#include <stdexcept>
...
   try {
     cout << v.at(10000000) << endl;
   }
   catch (out_of_range r) {
     cerr << "Range error:" << r.what() << endl;
   }
```

- `try` block contains statements that may raise an exception.
- `what()` returns null terminated char sequence that may be used to identify the exception.

When do exceptions get handled?

- `out_of_range` is a class.
- `throw out_of_range{"f"}` calls ctor with argument `"f"` that sets it's `what()` to return `"f"` and raises the exception.

```
void f() { throw out_of_range{"f"}; }
void g() { f(); }
void h() { g(); }

int main() {
   try { h(); }
   catch (out_of_range) { ... }
}
```

Sequence: `main` calls `h`, `h` calls `g`, `g` calls `f`, then `f` raises `out_of_range`.

Control goes back through the call chain (***unwinds the stack***) looking for a handler: checks `g`, `h`, then `main` where `main` handles the exception.

If no matching handler in entire chain $\Rightarrow$ program terminates.

Handlers can also throw exceptions

Multiple handlers can be part of the recovery job; a handler can execute some corrective code, then throw an exception to be caught by another handler.

```
try { ... }
catch (SomeErrorType s) { ... // partial recovery
  throw SomeOtherError{ ... };  // throw another exception
}
```

OR

```
try { ... }
catch (SomeErrorType s) { ...
  throw;  // throw the same exception
Alt: throw s;  // Not necessarily the same
}
```

`throw` vs `throw s`

Recall: `catch (SomeErrorType s) { ... }`

Suppose exception `s` is actually a type that is a subclass of `SomeErrorType`, rather than `SomeErrorType` itself.

- `throw s;` throws a new exception of type `SomeErrorType` slicing *s* into type `SomeErrorType`.
- `throw;` rethrows the actual exception that was caught, retaining its actual type.

Catch anything, throw anything

Can use `...` as a catch-all for exceptions.

```
try {
  . . .
}
catch (...) {  // literally use ... here
  . . .
}
```

Don't have to throw objects. Can throw anything.

- `exfact` and `exfib` (in repository) throw `int`s to compute factorial and Fibonacci.

Note: throwing exceptions is much slower than the recursive versions.

Define your own exceptions

Many existing exceptions, but you can also define your own exception classes for errors:

```
class BadInput {};

try {
  if (int n; !(cin >> n)) { throw BadInput{}; }
  catch (BadInput &) {
    cerr << "Input not well-formed\n";
  }
}
```

Note: exception caught by reference which prevents the exception from being sliced (if it's from a subclass of `BadInput`). Instead it will be treated like the kind of object that it actually is.

Catching exceptions by reference is usually the right thing to do.

Maxim in C++: Throw by value, catch by reference.

Other exceptions

NEVER let a dtor throw an exception!
By default, the program will terminate immediately by calling `std::terminate`. Also, if a dtor is being executed during stack unwinding, while dealing with another exception, and it throws an exception, there will be 2 active, unhandled exceptions and the program will about immediately.

Recall from early on: copy assignment operator for `Node`, Attempt #3

```
// If new fails, Node will still be in a valid state
```

When `new` fails, it throws the exception: `std::bad_alloc`.

Exception Safety

Consider the following:

```
void f() {
  C mc;
  C *p = new C;
  g();
  delete p;
}
```

Can this cause a memory leak?

Exception Safety

Consider the following:

```
void f() {
  C mc;
  C *p = new C;
  g();
  delete p;
}
```

Can this cause a memory leak? What if `g()` raises an exception?

What is guaranteed? During stack unwinding:

- All stack-allocated data is cleaned up as they are popped.
- dtors run for stack allocated objects.
- Stack memory reclaimed.
- **But heap allocated memory is not reclaimed!**
  $\Rightarrow$ if g throws, p is leaked.

Fix?

```
void f() {
  C mc;
  C *p = new C;
  try {
    g();
  }
  catch(...) {
    delete p;  // delete if g throws
    throw;
  }
  delete p;  // delete if g doesn't throw
}
```

- Duplication of code, often error-prone, etc

We want to guaranteed that in either case (exception or normal control flow) that certain code will execute at the end of our function.

How can we guarantee this?

Some languages allow a `finally` block (`try-catch-finally`) that contains code that is guaranteed to run, whether an exception was caught or not.

Some languages allow a `finally` block (`try-catch-finally`) that contains code that is guaranteed to run, whether an exception was caught or not. **But not C++.**

Only guarantee in C++ is:

- `dtors` for stack allocated objects will run.

$\Rightarrow$ use stack allocated objects with `dtors` as much as possible.
Use this guarantee to your advantage!

C++ Idiom: Resource Acquisition Is Initialization (RAII)

- Every resource should be wrapped in a stack-allocated object, whose job is to delete it.

RAII Example

For example: Files

```
{
  ifstream f{"sesame.txt"};
  ...
}
```

- Acquisition of the resource, the file, happens by initializing the object `f`.
- The file is guaranteed to be released (closed) when the stack frame `f` is allocated in is popped - `f`'s `dtor` runs.

Smart Pointers

This can also be done with dynamic memory using **smart pointers**.

- A template class that wraps raw pointers:
  `Class std::unique_ptr<T> // import <memory>;`
- Takes a `T *` (T pointer) in the `ctor`.
  The `dtor` will delete the pointer.
- Automatic memory management - automatically deallocate when they go out of scope.
- Dereference the object just like a normal pointer.

```
void f() {
  C c;
  std::unique_ptr<C> p{new C};
  g();
} // No leaks guaranteed!
```

Alternatively:

```
void f() {
  C c;
  auto p = std::make_unique<C>();
  g();
} // No leaks guaranteed!
```

- `ctor` arguments (if any) go in the parenthesis at the end.
- Allocates a `C` object on the heap and stores a pointer to it inside a `unique_ptr` object.
- `p`'s type is `std::unique_ptr<C>`

Example in the student repo.

Problems?

```
unique_ptr<C> p{new C{...}};
unique_ptr<C> q = p; // ERROR!
```

With pointer assignment, we think *copy the memory address*.

Problems?

```
unique_ptr<C> p{new C{...}};
unique_ptr<C> q = p; // ERROR!
```

With pointer assignment, we think *copy the memory address*.

But wrapped up in smart pointers is automatic deletion and we don't want to delete the same memory address twice.

So, can't simply copy the pointer!

Copying is disabled for `unique_ptrs`!

$\Rightarrow$ But they can be moved.

```
template <typename T> class unique_ptr {
  T *ptr;

public:
  explicit unique_ptr(T *p): ptr{p} {}
  ~unique_ptr() { delete ptr; }
  unique_ptr(const unique_ptr<T> &) = delete; // not available
  unique_ptr<T> &operator=(const unique_ptr<T> &) = delete;

  unique_ptr(unique_ptr<T> &&other): ptr{other.ptr}
    { other.ptr = nullptr; }

  unique_ptr<T> &operator=(unique_ptr<T> &&other) {
    std::swap(ptr, other.ptr);
    return this;
  }

  T &operator*() { return *ptr; }
};
```

Design Questions

If you need to be able to copy pointers, first answer the question of ownership:

- Who will own the resource?
  Who will be responsible to free it?

- That pointer should be a `unique_ptr`.
  All other pointers shoud be *raw pointers*.
  $\Rightarrow$ The `unique_ptr` will automatically delete the object and no need to delete *raw pointers*.

If needed, you can fetch the underlying raw pointer using `p.get()`.