ECE 2574: Data Structures and Algorithms - Error Handling and Exceptions

C. L. Wyatt

Today we will look at more detail into error handling mechanisms and the use of exceptions.

- Motivation: what can go wrong?
- The two and a half approaches to error handling
- Defining and returning error codes
- What is an expected error and what is *exceptional*
- exceptions and try/catch
- using `std::except`
- exception-safety

Some operations that can fail

- opening files for reading and writing

```
std::ifstream ifs("afile.txt");
// what if afile.txt does not exist?
```

- parsing: example

```
int value = std::stoi("123!");
```

- memory allocation

```
std::vector<int> data(1000);
// what if allocation fails inside vector?
```

For each line of code you write, think: *How could this fail*?

Approaches to error handling

- none (bad)
- static error flags
- returning errors
- object-specific error status
- exceptions

Error status flags

This is common in some older C (and C++) libraries.
Store a static error code that can be queried.
See example code: `static.cpp`
This has problems with concurrency and litters the code with checks

returning error codes

This is similar to error flags but the code is returned

- return as an output argument. See example code coderet1.cpp
- use a std::pair to return multiple arguments, one of which is the error code. See example code coderet2.cpp
- Use std::tie to unpack the returned std::pair. See example code coderet3.cpp

This still litters the code with checks, but is reasonable.

C++17 has std::optional which will give us an even cleaner approach

See example retopt.cpp

object specific error

You can store the error status internal to the object an query it.
See example code: `objecterr.cpp`
For example, this is how the std stream interface works, e.g.

```
std::ifstream ifs("afile.txt");
if( ifs.good() ) ...
```

This is a good approach to common error conditions. Can be adapted to concurrent programming.

Exceptions

Exceptions interrupt the normal flow of a program.
When an exception is *thrown*, execution continues at the previous function call that *catches* the exception.
This continues until a previous caller catches it, or your program aborts.
See examples: `exception1.cpp` and `exception2.cpp`.

Using std::exception

If you define your own exceptions, derive them from std::exception
See example `derivedexept.cpp`

Good things about Exceptions

- They make the code much cleaner.
- They reduce the number of sequential checks that go on.

This makes it easier to see the non-error handling logic.
They also allow you to handle errors where you can (sometimes) do something about it.

Problems with Exceptions

- They complicate resource management. See example `exception3.cpp`.
- They can be somewhat hard to reason about.
- They make binaries larger

Overall though exceptions are worth the effort.
However, be sure they are exceptions. Never use exceptions for normal program flow.
All errors are not exceptions, in particular anything that is triggered by user input.

How to handle errors

- handle it if you can, retry a network connection for example
- Notify the user
- Cleanup resources
- Die with dignity.

special considerations for memory allocation failures

If you are out of memory anything in the exception handling code cannot allocate!
This includes things like std::string.

Exception specifiers

It is possible in C++ to say that a method may throw an exception.

```
void mymethod throw(A,B)
```

but it has never worked well (for reasons to detailed to go into here). My recommendation is to not use them, but instead document possible exceptions in the comment for the method.

Exception safety

Related to exception specifiers is the `noexcept` keyword, which says that a function does not throw an exception.
This is good practice **if** you can guarantee it will not.
Such code is called *exception-safe*.

Larger Example

Parse a IP address in the form 192.168.0.1 into a 32 bit integer.

- Version using exceptions: `parse_with_exceptions.cpp`

Next Actions and Reminders

- Read CH chapter 8 (List ADT)
- Complete the Warmup before noon on Wednesday 10/4.