COSC 2P95

Error checking and debugging

Week 10

Brock University

Reliability

We've spent several weeks now learning to write solutions, using various different tools and approaches.

However, outside of minimal *sanity checking*, we've had to mostly rely on inputs/datas/solutions that "just work".

- That really isn't plausible for a complete system of any significant size

There are a few basic concerns when developing solutions:

- Code must be written 'correct' to the language
- Algorithms should be free of logic errors
- Never trust a human
  - Never trust a machine, either, if you can avoid it
- When things go wrong (and they will):
  - Recovering is nice
  - *Detecting* the problem is paramount

Syntax vs semantics

Before we get any further, we first need to ensure that we remember the difference between *syntax* and *semantics* — and, by extension, syntax errors vs semantic errors

- Forgetting a semicolon? That's a ~~paddlin'~~ syntax error
- Using the wrong index, or calculating a formula wrong? Semantics!
- Generally, this is the difference between compile-time and runtime
  - Rule of thumb: compile errors are no big deal. Finding out something's broken after you've deployed it? Not pleasant

What actually goes wrong?

What do we do about it?

Oftentimes, outside of pure logic/algorithm errors, semantic errors are a result of *violated assumptions*.

- e.g. you tried reading from a stream; since it returned, you assume it didn't use a default 0
- Or you asked the user for a number from 1 to 10, and *assumed* they wouldn't enter "cheeseburger"

What can we do about it? Well, the first step is *defensive programming*.

- Don't write loops that end on zero, if you could end on ≤ 0
- For prompts for user input, consider putting them in loops until they receive legal input, or include a 'default behaviour'
- Check status flags of streams after access

However, we can still find ourselves making assumptions. Can we verify them?

Assertions

A common debugging technique is to add dozens of *print* statements, to report *happiness messages*.

- If the code got to a certain point, it prints
- If a value contains what you expected, it prints

But, for any significant quantity of tests, that quickly becomes impossible to sift through. There's an alternative: *assertions*. e.g.:

```
#include <cassert>
...
int value;
std::cout<<"Give a number from 1 to 5: ";
std::cin>>value;
assert(value>=1 && value<=5);
std::cout<<"Good choice!"<<std::endl;
```

Let's try running this to see what happens.

Better assertions

Sometimes, you might want to either define a `bool` for your test, or use the assert to check for *non-null*. For these cases, even if it failed, you'd end up with an output equivalent to 'infile' failed.

- If desired, you can write something like:
  `assert(infile && "Much more descriptive feedback");`
  - The status of `infile` is what will dictate the result

Great! I'm done!

Should I keep the testing code in?

This may sound counterintuitive, but you *don't* want the assertions in your final release.

- The typical end-user won't be able to do anything with the error messages anyway
- The errors might not be catastrophic, but the program *won't* execute past tripped assertions
- Even with the previous slide's suggestion, the feedback typically isn't adequate for proper troubleshooting
- Though bugs are still possible, by the time you get to production, you should be confident that you've removed the major bugs that would have triggered your assertions
  - Things like "I hope I understand how this formula works" and "my file-loading function actually loads files" *should* be pretty reliable

If you add the preprocessor directive `#define NDEBUG` *before* including `cassert`, you turn them off.

- It's also slightly more efficient

So, now my code is done?

I can trust its operation?

- *Ah ha ha ha ha ha ha*

… really?

…ha ha ha ha ha ha ha!

Stoppit

Sorry.

No, we still have several concerns:

- Even completely correct code will sometimes encounter bad situations
  - Maybe we can recover; maybe not. But we need to identify them
  - We may need to handle such issues at a different level from the local context where they're encountered
- Some issues (entering a number outside the required range) can be immediately addressed, but how do we tell a calling context that there was an issue inside a function?
  - Return a `bool` type? Yes, sometimes
  - Include a reference parameter for a status flag? Possibly, but that gets nasty for multiple possible errors
  - For classes, add a status flag? Sure, if you're already using a class, and changing the specification is okay

Basically, we have a *lot* of other cases. What we really need is the ability to flag an issue, suspend execution, and immediately jump straight to code intended to handle that issue.

Exceptional events

C++ offers a tool to us: the *exception*.

- We're *not* going to learn *goto* statements in this course, but as a thought experiment, suppose your program could respond to a problem by *immediately* dropping everything, and *jump*ing straight to a special block of code, dedicated to handling that problem
- If that problem's never encountered, that jump never occurs, and that special code is never used

We *do* have a mechanism for this.

Exceptions

`throw`

In C++, we can `throw` whatever we like. An `int`, a `float`, a `Monkey` (but ignore that one for now).

- When something is *thrown*, execution at the current position in the code *immediately stops*
- We'll get into more detail in just a moment, but basically the program tries to find a *handler* capable of receiving whatever was thrown
- Depending on the situation, either the fact that *anything* was thrown, or precisely *what* was thrown will indicate the problem, and the program will respond appropriately
  - Responding could mean correcting the issue, or asking the user for additional input, or it could simply trigger a *graceful exit* of the program (closing streams, etc., and possibly displaying a message or error code)

Exceptions

`try/catch`

Simply throwing something wouldn't have any use. Instead, we'll *normally* want to tell the system to get ready for it.

- We put code that we anticipate potentially triggering an *exceptional event* inside a `try` block
- We follow a `try` block with a `catch` block
  - The `catch` block is matched up to the same type as what's thrown, and receives the thrown value
  - The `catch` block is where you handle that exceptional event
- A single `try` block may have *multiple* `catch` blocks
  - Each `catch` block is matched to a different thrown type
    - If you want to throw, for example, multiple `int` types, you'll need a condition in your `catch` block

Can you throw a value *without* a `try/catch`? Yes, but it isn't advisable. Example time!

Exceptions

Different contexts/call frames

Of course, it would be ridiculous if exceptions needed to be thrown within the same context as their try/catch blocks.

- How would that be significantly different from any other regular conditional?

The reason the program terminates without a try/catch is because the exception is actually trying to *propagate upwards/outwards*.

- If the exception can't be caught in the current context/function, then it immediately leaves to the calling context and tries there
  - And if *that* context can't receive the exception either, then it continues outwards
  - Since there's nothing above/outside of `main`, that's when it terminates

Exceptions
Stack unwinding

That process of leaving the current context, to revert back to the calling context, is called *stack unwinding*.

A bit cliché, but it's not a bug; it's a feature.

If there *is* a catch block in one of the calling contexts, then it can be handled *there*, instead.

- This is both important and valuable, because a local context might not be equipped to fix the problem

This seems like a good place for another example.

Exceptions

Catch-all handlers

That last example was missing a single handler.

- Oftentimes, you'll encounter circumstances from which there really is no recovery
  - Technically, this doesn't *have* to be one of those cases; even after an EOF, you can reset the stream, but that would be contrary to the usage intent
- This presents two special considerations:
  - We probably can't do anything other than terminate
    - But we should still *exit gracefully*
  - Whatever we choose to do, something like a complete loss of user input should possibly be handled by the `main`
    - But if we start deferring *all* of our 'special cases' to main, we'll eventually need to stop looking for specific cases, particularly if we reach this point because we just need to exit gracefully (irrespective of what the exception is)
- There's a *catch-all* handler: . . .

Let's fix our example, shall we?

Fancier exceptions

As mentioned earlier, you can `throw` whatever you need to throw. If you need to throw a monkey, throw a monkey.

But, what if you want better feedback than simply throwing an `int`, or a `double`, or even a string, but there's no existing data type that satisfies that need?

- You can define a new class, solely for the purpose of passing along such an exception message, if you wish
- Typically, you'd at least include some form of flag or string message, to hint at what went wrong

Besides the included message/data, simply being a uniquely-chosen type can provide information.

- For example, if you create a `DivisionByZero`, then that's enough to know what happened
  - Though, if desired, you can still also include member variables for the numerator/denominator

Fancier exceptions

Something to be careful about

Note that, just as with any other classes, we *can* include inheritance here as well. e.g.:

- `MathError`
  - `DivideByZeroError`
    - `ZeroByZeroError` (because we like feeling special)
  - `NegativeRootError`

If you do this, then be *very* careful of the sequence in which you list the `catch` blocks.

```
try {/*code*/}
catch (DivideByZeroError) {/*code*/}
catch (ZeroByZeroError) {/*code*/}
catch (NegativeRootError) {/*code*/}
catch (MathError) {/*code*/}
```

Because `MathError` is listed last, `DivideByZeroError` and `NegativeRootError` can trigger, but `ZeroByZeroError` *cannot*, as it's listed *after* `DivideByZeroError`!

Fancier exceptions

Standard library

It's worth noting the standard library (under the `exception` header) provides for a hierarchy of exceptions.

- The `exception` parent class isn't terribly interesting, but its subtypes are worth looking at (or define your own)
  - e.g. `runtime_error` accepts a message, to be later retrieved via `.what()`

`http://en.cppreference.com/w/cpp/error/exception`

Final thoughts on exceptions

Honestly, exceptions can be pretty darn handy, but keep a few things in mind:

- They aren't strictly necessary
  - For larger, object-oriented systems, you'll probably want a versatile system for error-checking, but for smaller (especially purely procedural) applications (e.g. number-crunching), you might want to skip them
- Be *very* careful about things like resource management
  - Closing streams, releasing memory, etc.
  - There *are* some fancy pointer templates that can help with this
- If you aren't catching the exceptions you're throwing, one might wonder what the point is
- Using exception handling to mask catastrophic errors is like sticking electrical tape over the *check engine* light

Additional tips

Keep in mind that there are *many* ways to track down persistent bugs.

- There's a Gnu Debugger (GDB), if you're interested in command-line debugging
- Valgrind can be a particularly useful tool for tracking down memory issues
  - Are you getting inconsistent segfaults? You should check out Valgrind
- Most IDEs allow you to add things like breakpoints, or otherwise aid with debugging

And, of course, the best advice of all for eliminating bugs:

- A well-designed solution with no logic errors to begin with is less work than trying to fix spaghetti

Questions?

Comments?

- Funky tunes?