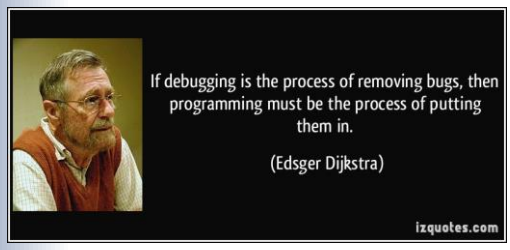
CSCI 262
Data Structures

18 – Debugging

CS@Mines

Debugging



CS@Mines

Strategies and Tools

- Develop incrementally
- Watch for familiar bugs
- Divide and conquer
- Use `cout` to expose inner workings of code
- Instrument your code
- Design custom inputs
- Focus on recent changes
- Understand error messages
- Use a debugger

CS@Mines

Incremental Development

Think before you code.

- Break project down into small tasks
- Break tasks down into small functional chunks
  - E.g., task is "read in file":
    - Obtain filename
    - Open file/check for errors
    - Read in lines
- For each chunk:
  - Comment the bit of code for that chunk
  - Write the bit of code
  - Test the bit of code and get it working
  - Only then go on to the next chunk!

CS@Mines

Watch for Familiar Bugs

Some common mistakes:

- "=" instead of "==" (or vice versa)
- "Off by 1" errors (in loops, array indices, etc.)

```
for (int j = vec.size(); j > 0; j--) { vec[j] = 0; }
```

- Scoping errors:

```
int j = 0;
while (j < 100) {
    int j;
    j++;
}
```

- Not incrementing the loop counter (my personal bug)
- Copy/paste bugs

```
for (int i = 0; i < foo; i++) {
    for (int j = 0; j < bar; i++) { … }}
```

CS@Mines

Off-by-One

"There are only two hard things in computer science: cache invalidation, naming things, and off-by-one errors."

CS@Mines

Divide and Conquer

Iteratively find the smallest code subset that exhibits a bug:

- Start by commenting out roughly half of your code
- If bug disappears, uncomment, comment other half
- Now split the remainder in half again, etc.

Isolating *where* the bug occurs is essential, especially when bugs are subtle.

Use cout to Expose Inner Workings of Code

- Do you know what your code is up to?
  - Code is a black box
  - Difficult sometimes to reason through logic
  - Find out what is happening by opening the box!
- Works with most other strategies
- Alternative to debugger

cout

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."

- Brian Kernighan

Instrument Your Code

- Add "probes" to your code:
  - Use `if` statements to test critical values
  - Output error messages on failure of expectations
  - Probes are also good sites for debugger breakpoints!
- Collect statistics – compare with expectations
  - How many times is your loop *actually* running?
  - What was the average color value processed?
  - What are my current point coordinates?

Probe Example

```
while (condition) {
    // probe: check for invalid index
    if (j + 1 >= vec.size()) {
        cout << "Oops!" << endl;
    }

    // code breaks somewhere in here
    vec[j + 1] = …
}
```


Design Custom Inputs

- Create small/simple inputs for your problem
  - Small data file
  - Easy/trivial input
- Works well with:
  - Instrumentation
  - cout
  - Debugger

Focus on Recent Changes

- Did it break?
- Did you mess with it?
  - Look at the changes you made, first
  - Bug could still be elsewhere in code ☹

CS@Mines

Understanding Error Messages

- Build errors
  - Build errors (especially in C++) can be cryptic
  - Incremental development helps a lot!
    - Fewer error messages to dig through
    - Less code likely to have caused error message
  - Use Google!
- Runtime errors
  - Segmentation fault, aka "segfault"
  - Exceptions

CS@Mines

Debugger

- Lets you set breakpoints (function/line to start debugging) optionally with conditions
- Lets you step through code one line at a time
  - Over/into/out of functions
  - Continue to next breakpoint
- Stops execution on exceptions, segfaults
- Lets you look inside your program:
  - Print/display values of active (in scope) variables
  - Backtrace (display stack frames – all active functions and their argument values)

Debugger + patience = almost certain success!

CS@Mines

Debugger Demo

CS@Mines

Up Next

- THANKSGIVING BREAK – Have a great break!
- Monday, November 26
  - Go over midterm 2
- Still to come:
  - Hashtables
  - Inheritance
  - One other topics (TBA)

CS@Mines