CSCI 262
Data Structures
12 – Testing and Debugging

Goals for this Lecture
• Learn common strategies and tools for
  – Writing code
  – Testing
  – and Debugging
• In order to reduce errors (and improve grades!)

Testing vs Debugging
• Edsger W. Dijkstra:
  “Program testing can be used to show the presence
  of bugs, but never to show their absence!”
• Testing is trying to break your program
• Debugging is trying to fix your program

Strategies and Tools
• Incremental development
• Watch for familiar bugs
• Divide and conquer
• Instrument your code
• Design custom inputs
• Use cout to expose inner workings of code
• Use a debugger
• Focus on recent changes
• Understanding compile time errors

Incremental Development
• Break project down into small tasks
• Break tasks down into small functional chunks
  – E.g., “read in file”
  • Obtain filename
  • Open file/check for errors
  • Read in one line
  • Read in all lines
• For each chunk:
  – Comment the bit of code for that chunk
  – Write the bit of code
  – Test the bit of code
  – Only then go on to the next chunk!
  Think before you code.

Watch for Familiar Bugs
Some of my common mistakes:
  – “a” instead of “==”
  – Not incrementing the loop counter
    ```cpp
    while (j < some_number) {
      // infinite loop because I don't increment j
    }
    ```
  – “Off by 1” errors (in loops, array indices, etc.)
    ```cpp
    int index = rand() % list.size() + 1;
    ```
    ```cpp
    for (int j = 0; j < some_number; j++) {
      // do something
    }
    return foo(j);
    ```
  – Copy/paste bugs

*There are only two hard things in computer science: cache invalidation, naming things, and off-by-one errors.*
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 the function where the bug occurs can be very helpful, especially when bugs are subtle.

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?

Design Custom Inputs
• Your code is doing something funny:
  - Your image is the wrong color
  - Your lab solution is consistently off by a little bit
• Create a custom input that should behave in a certain way (if the bug were not there):
  - Create an image that is all blue
  - Create a simple valid input for your lab
• Make more complex inputs if a simple one doesn’t reveal the bug
• Can work with probes:
  - Use if statements to check values when a specific input is encountered

Use cout to Expose Inner Workings of Code
• Strategy: use cout << to display variables, statistics, etc. throughout your code
• This strategy works together with most of the others
  - cout probe sites
  - cout expected behaviors for custom inputs
• Alternative to using a debugger

Debugger
• Powerful, but potentially tedious
• Lets you step through code one line at a time
• Lets you set breakpoints (what line to start debugging on)
• Sometimes, lets you stop execution when exceptions occur
• Gives you diagnostics:
  - Print/display values of all active (in scope) variables
  - Backtrace (display of your stack frames – all active functions and their argument values)
• In the worst cases, the debugger and lots of patience may be your only hope!

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 😊
Understanding Build Errors

• Build errors (especially in C++, especially in VS) can be cryptic
• Incremental development, divide and conquer strategies help a lot with focusing on the specific error
• Use Google!