Debugging

*CIS*2750*

Advanced Programming Concepts

*Some material for this lecture has been taken from **Code Complete** by Steve McConnell*

What is Debugging?

- Debugging is the process of *identifying the cause* of an error and *correcting it*.
  - *Not* a way to improve **software quality**
- View debugging as a last resort
  - Try to develop programming habits that greatly *reduce* the need to find errors.
- Spend the time observing and analyzing the errors that you do make…
  - Discover ways to avoid making them in the future!

Debugging Performance

- Studies show:
  - **20-to-1** difference in the time it takes for an *experienced* programmer compared to an *inexperienced* programmer to find the same set of errors!
- You can develop better debugging techniques over time!

Debugging Performance

- Develop a set of standard approaches to finding errors and keep the ones that work more often then those that do not.
- When you encounter an error and then solve the problem, write down what you did in a file that you keep especially for debugging.
- Over time you will develop your own set of **best practices**.

Errors as Opportunities

- Learn about the program you're working on.
  - Is the origin of the defect in the **requirements**, **specifications**, **design** or **implementation**?
- Learn about the *kind* of mistakes you make.
  - Can one (bitter) debugging experience help to eliminate future defects of a similar nature?

Errors as Opportunities

- Learn about the quality of your code from the point of view of someone who has to read it.
  - The ability to *read* programs is not a well developed skill in most programmers.
  - The result is also the inability to write *readable* code.
- Learn about how *you* solve problems.
  - Observing and analyzing how you debug can *decrease* the total amount of time that it takes the next time you develop a program.

Errors as Opportunities

- Learn about how you fix errors.
  - You should strive to make *systematic* corrections.
  - This demands an accurate diagnosis of the problem and an appropriate prescription that attacks the *root cause* of the defect.

The Devil's Guide to Debugging

- Find the error by guessing!
  - Scatter print statements randomly throughout the code.
  - If the print statements do not reveal the error, start making changes until something appears to work.
  - Do not save the original version of the code and do not keep a record of the changes that have been made.
- Debugging by superstition
  - Why blame yourself when you can blame the computer, the operating system, the compiler, the data, other programmers (especially those ones who write library routines!), and best of all, the stupid users!

The Devil's Guide to Debugging

- Don't waste time trying to understand the problem.
  - Why spend an hour analyzing the problem (in your head and on paper) and evaluating possible solutions or methodologies, when you can spend days trying to debug your code?
- Fix the error with the most obvious fix.
  - For example, why try to understand why a particular case is not handled by a supposedly general subroutine when you can make a quick fix?
- Show someone an *old* version and ask for help.

The Scientific Method of Debugging

- Gather data through repeatable *experiments*.
  - Stabilize the error, narrowing possible explanations.
- Form a *hypothesis* that accounts for as much of the relevant data as possible.
  - Locate what seems to be the source of the error.

The Scientific Method of Debugging

- Design another experiment to prove or disprove the hypothesis.
- Fix the error according to hypothesis, then test the fix.
- Repeat as needed.
  - Look for similar errors.

Tips for Finding Errors

- Use all the data available to make your hypothesis.
- Refine the test cases that produce the error.
  - Reproducing the error several ways helps to diagnose the cause of the error.
  - Errors often arise from a combination of factors; *one* test case may not be enough to find the root of the problem.

Tips for Finding Errors

- Generate more data to generate more hypotheses.
  - Run more test cases to help in hypothesis development and refinement.
- Use the results of *negative* tests.
  - Negative results can eliminate parts of your search space.
  - You have gained more knowledge about the program and the problem.

Tips for Finding Errors

- Brainstorm for possible hypotheses.
  - Do not limit yourself to just one hypothesis; generate several.
  - Do not limit the problem solvers to just yourself; *other people* can bring new perspectives and experience to your problem.
  - Concentration on a single line of reasoning can result in a mental logjam.

Tips for Finding Errors

- Narrow the suspicious region of code.
  - Systematically eliminate parts of your program to isolate the part that contains the defect.
  - Yet another argument for modularization!
- Be suspicious of modules that have had errors before (may have poor code quality).
  - Re-examine error-prone modules.
- Check code that has recently been changed.
  - Compare the old and new versions of your code.

Tips for Finding Errors

- Expand the suspicious region of the code.
  - Do not focus on a narrow piece of the system, even if you are sure that the error must be in that particular section.
- Integrate incrementally.
  - Add pieces to the system, one at a time.
  - After each addition, test the system to detect errors.
  - If there are errors, you know the area to focus on!

Tips for Finding Errors

- Set a maximum time limit for quick and dirty debugging.
  - If you cannot find the error within your time limit, then it has to be admitted that the error is a hard one requiring experimentation/research.
- Check for common errors.
  - Use code-quality checklists to stimulate your thinking about possible errors.
    - *see **Code Complete** by Steve McConnell*

Tips for Finding Errors

- Talk to someone else about the problem.
  - *Confessional debugging* can help even by just forcing you to organize your thoughts about the problem in a way that someone else can understand. This process may be enough for you to arrive at the solution yourself without any *input* from others.
- Take a break from the problem.
  - The subconscious can be a great problem solver and, of course, food, sleep and sunlight are necessary for sustaining life functions.
  - If you have *debugging anxiety*, stop and take a break.

Syntax Errors

- Do not fully trust compiler messages:
  - The line number could be off (by 1)
  - Second messages can come from an *earlier* problem, not the one it's claiming now
- Never completely trust any software tool
  - Always understand the tool (gdb, valgrind)
  - The more you know about *software tool x*, the more use it can be (and avoid misapplying it)

Syntax Errors

- Divide and conquer!
  - Separate out parts of the code and run it through the compiler to isolate the syntax error and/or get a more reasonable error message from the compiler.
- Study and understand the syntax errors peculiar to the programming language that you are using.

Memory Management Errors

- Abuses that valgrind detects:
  - use of **uninitialized** variable or **invalid read** (where no storage allocated)
    - segfault (dereference NULL/bogus pointer) or other exception (divide by 0)
    - if/loop condition decided based on bogus value
      - infinite loop possible (hangs, needs ^C interrupt)
    - calculation based on bogus value
    - string is taken as wrong length
  - **invalid write** (where no storage allocated)
    - corrupts some variable
    - corrupt value may cause above problems *later* (time bomb)

Runtime Effects Vary and Deceive: 3 Random Effects

1. Abuse is *not noticed* ("it works perfectly!")
   - You got lucky!
     - bogus value read was not harmful in context (say, 0)
     - corruption didn't damage something important
2. Dynamic memory manager (glibc) noticed!
   - Glibc error or dump → print/abort
   - Library assertion fails → print/abort
3. Mem. manager or your code crashes
   - Segfault or infinite loop "hangs"

Glibc Behaviour Also Varies

- MALLOC_CHECK_ environment variable
  - 4 different levels of detection/treatment
    - unset, 0, 1 (default), 2, 3 //use 1 for debugging
- Same bug can manifest in any of previous 3 ways depending on setting!
- Automarker woes:
  - We have to choose *a* setting → permutes which people get which effects from their mem. abuse

“Getting Lucky” Is Not the Goal!

- Robust/reliable software is!
- *Tips:* As soon as you see…
  - a mysterious crash, run in the debugger!
    - Stops at crash, show **backtrace**, examine variables.
  - erratic behaviour, run in valgrind, stamp out every error! (invalid write especially evil)
    - Take hint from **size** of read/write (typical 64-bit):
    - 1 = char or string length problem, 4 int, 8 pointer

What About Memory Leaks?

- If you don't call free(), you have leaks:
  - Sign of sloppy, unprofessional coding!
  - Program that needs lots of memory and/or long run time can exhaust memory → crash.
- If you do call free(), you *could* have leaks:
  - Sign of logic error! Call wasn't executed.
  - Finding/fixing logic errors is good! Often involves code *near* the malloc/free calls, which you only noticed from tracking down the leak.

Fixing an Error

- Understand the problem before you fix it.
- Understand the *program*, not just the *problem*.
  - Study done with short programs:

    Programmers who achieve a *global* understanding of program behaviour have a better chance of modifying it successfully than those who focus on *local* behaviour, learning about the program only as they need to.
  - A large program may not be understood in total, but the code in the vicinity (a few hundred lines) of the error should be understood.

Fixing an Error

- Confirm the error *diagnosis*.
- Relax -- Never debug standing up.
- Save the original code (source control!)
  - It is easy to forget which change in a group of changes is the most significant.
  - It is always useful to be able to compare the old and new versions of code to verify all changes.
- Remember that defect corrections have more than a 50 percent chance of being wrong the first time!

Fixing an Error

- Fix the problem, not the *symptom*.
  - If you do not understand the problem, then you are not fixing the code -- you are fixing the symptom and could be making the code worse.
  - Problems with fixing symptoms include:
    - The fixes will not work much of the time.
    - The system will become unmaintainable.
      - Too many special cases!
    - It is a misuse of computers -- understanding the problem involves the programmer not the computer.

Fixing an Error

- Change the code only for good reasons.
  - *"Never change code that works."* Why?
- Make one change at a time.
- Check your fix -- do regression testing.
- Look for "similar" errors.
  - If you cannot figure out how to look for similar errors, then you probably do not completely understand the problem.