How To Debug Programs, Part 1
General Debugging Approaches

CS24 – Spring 2011

CS24 and Debugging

- Debugging is an essential skill for all programming
- Particularly important in CS24:
  - Assignments involve relatively complex systems, frequently with multiple moving parts
- May also be the first class where you must debug a binary program, not something running in an interpreter
  - Must use other tools to peer inside this black box as it runs
  - Tools may be confusing, but you <u>must</u> learn them and use them
- Knowing how to debug problems effectively pays off:
  - For CS24, it can easily cut <u>hours</u> off of your assignments
  - If you end up programming for a living, knowing how to debug well will make you a superstar programmer

CS24 and Debugging (2)

- These lectures are to help you learn how to debug better
  - Note: there are many other considerations when you debug in a professional software development environment
  - This lecture is mainly to help you fix your own bugs
- Part 1: What are the basic principles and approaches of debugging?
  - [Mostly] independent of specific language, platform or toolset!
- Part 2: What tools and approaches can I use to debug my C and IA32 assembly language programs in CS24?
  - GDB, Valgrind

Bugs

- What is a "bug" anyway?
  - Helpful to break into two different components
- The program contains a <u>defect</u> in the code, possibly due to a design issue or an implementation issue
- This defect is manifested as a <u>failure</u> of some kind
  - Program produces an incorrect result, or it crashes, etc.
- Defects are not always manifested:
  - May require specific data inputs to produce the failure
  - May require running the program on a specific platform
- Defects are also not always manifested immediately!
  - May require running a program for hours, days, or weeks (!!!) before the failure occurs

Finding Bugs

- The majority of the work in fixing a bug is finding the actual defect that produces it
  - Usually, once the defective code is identified, a fix is very easy!
  - If overall *design* is defective, this can be much more difficult: can require redesigning and reimplementing a large portion of the program
- The defect is only the *cause* of the failure, but it is not the failure itself!
  - The defect will immediately begin to affect the program's state, but the effects may not become visible for some time
  - The greater the separation between defect and failure, the harder it is to diagnose the defect from the failure

Finding Bugs (2)

- If the defect directly causes the failure:
  - e.g. your loop's logic is broken and it dereferences a NULL ptr
  - Happy days! Simply need to identify point that the program fails, and can fix the bug very quickly
- Often, the defect and the failure are separated by a significant amount of execution time
- Example:
  - Function f1() mangles a linked list, but completes successfully
  - Function f2() attempts to use the linked list, and fails miserably
- In this case, the bug itself is not actually in f2()!
  - The defect is in f1(); its *manifestation* is in f2()
- How do we determine the actual <u>cause</u> of the failure?

Preemptive Bug Detection

- One common technique for causing defects to manifest quickly is using <u>assertions</u>
  - Frequently have conditions that you expect to be true at certain points in your program
  - Explicitly state these in an assertion, in your code
  - At runtime, the assertion is checked: if it's false, the program is stopped immediately!
- In C programs:
  - #include <assert.h>
  - assert(*condition expected to be true*);
- Use assertions to check function arguments, return values, and the state managed by your functions
  - See both the provided code and solution sets for examples!

Detective Work

- Once you have observed a failure in your program, you have a mystery to solve!
- One thing that no good detective ever does:
  - Guess randomly!
- The programming version of this:
  - Make random guesses as to the cause, and try various changes
  - Called "the shotgun approach," and "monkeys on a typewriter"
  - (The results are just as good, too.)
- Considering various clues, you must track down the defect from the indications of failure
- Once the issue is identified, *then* make your fix
  - Not as likely to introduce other defects this way, too.
  - (Hunt bugs with a rifle, not with a shotgun.)

Step 1: Reproduce the Failure

- Before you can do anything else, you <u>must</u> find a way to reproduce the failure
  - Recreate the steps that caused the program to fail
  - Was it specific data inputs? Was it a specific interaction with the user interface? Does the program fail under heavy load?
- This is very important, for three major reasons:
  1. **So you can <u>watch</u> it fail.** You can see exactly what the program was doing as it crashed and burned.
  2. **So you can zero in on the cause.** If the program fails in some circumstances but not in others, this will give you *hints* as to what part of the program actually contains the defect.
  3. **So you can test if you actually fixed it.** If you can <u>reliably</u> cause the failure, and your fix makes it go away, you win!

Step 1: Reproduce the Failure (2)

- Which of these reasons actually requires debugging tools?
  1. **So you can watch it fail.** You can see exactly what the program was doing as it crashed and burned.
  2. **So you can zero in on the cause.** If the program fails in some circumstances but not in others, this will give you *hints* as to what part of the program actually contains the defect.
  3. **So you can test if you actually fixed it.** If you can reliably cause the failure, and your fix makes it go away, you win!
- Debugging tools and techniques allow you to watch your programs fail
  - …or to see what they *were* doing when they failed…
  - A small but very critical part of the challenge of debugging

Technique: Keep a Debugging Log!

- When debugging, it's extremely helpful to keep a record of your efforts (hand-written or typed, it doesn't matter)
  - A general description of the failure
  - Inputs or circumstances in which the failure occurs
  - Ideas of potential causes, along with the efforts you made to verify your ideas, and indications for or against each theory
- Very effective for helping focus your thoughts and ideas
- Also allows you to set aside an issue, and pick it up later
- It's generally a very bad idea to debug while exhausted
  - Thought processes aren't clear; can't reason about the bug
  - Tend to revert to "monkeys on a typewriter" mode
  - Put it down, walk away; come back later when you're fresh

Step 2: Isolate the Failure

- So your program fails. And you know how to make it fail.
- Problem: the defect is only in a small part of your code
  - Need to zero in on the part of the code that's actually flawed
- Goal: try to devise the smallest possible scenario that still causes the failure to occur
  - If the scenario is small, there won't be very much code involved
  - The less code that's involved, the easier it is to find the defect

- Example: a program that processes large log files
  - A particular log file causes it to crash. The log file is 27MB.
  - What do you do?

Example: Failure Isolation

- Example: a program that processes large log files
  - A particular log file causes it to crash. The log file is 27MB.
  - What do you do?
- If you have the offending log file, can reproduce the failure
  - Still, a lot of code is executed before the failure occurs
- Want to narrow in on the actual cause of the bug:
  - Cut the log file down until you have the *minimal* portion that still causes the crash
  - Perhaps the program crashes because log contains bad values
  - If you're lucky, may identify a handful of input lines that causes the problem
  - Should make it much easier to identify and resolve the defect

Example: Failure Isolation (2)

- Example: a program that processes large log files
  - A particular log file causes it to crash. The log file is 27MB.
- What if you can't get the issue to reproduce by trying smaller parts of the log file in isolation?
- That would be a much harder bug to track down…
  - …but, would indicate that the bug is triggered by some characteristic of the log file *as a whole*
  - e.g. maybe log's size causes some memory management issue
  - e.g. maybe the combination of some values read from the log causes the failure
- You learn more about the nature of the defect by trying to isolate it! (Record these things in your debugging log!)

Step 2: Isolate the Failure (2)

- Previous example involved trying to isolate the *inputs* that cause the failure
- Can also try to isolate the general part of your program's code that may cause the failure

- Example: memory allocator in assignment 3 ☺
  - When you run the allocator testing program, it crashes
  - Easy to reproduce: every time you run it, it crashes
  - But, the testing program doesn't have any inputs.
  - What do you do?

Technique: Understand the System!

- Knowing where to start with this bug really requires you to understand the system being debugged. For example:
- What are the major functional components of program?
  - What portions of the code perform each of these functions?
  - If failure manifests when a specific feature is used, you know what files to start focusing on.
- Are there portions of the code that are <u>always</u> executed?
  - If failure manifests in a range of different scenarios, this would be a likely location for the defect.
- Are there portions of the code that can easily be disabled or removed, and the test run again?
  - If part of the code can be removed and the failure still occurs, we know the cause is not in the code that was removed.

Technique: Understand the System! (2)

- Can ask similar questions for the inputs to your programs
- Are there portions of the code that only execute with specific data inputs, or sequences of inputs?
  - e.g. are certain parts of your program executed only when you feed the program specific inputs?
- Example: SparseVector in CS11 C++ track
  - Represented as a linked-list of elements, kept in order of index
  - By feeding in specific sequences, can target testing more specifically
  - Add values to the vector with decreasing indexes:
    - Tests linked-list prepend code
  - Add values to the vector with increasing indexes:
    - Tests linked-list append code
  - Add values to the vector with varying indexes:
    - Tests linked-list insert code
  - Each of these is usually implemented in a *different* piece of code

Technique: Understand the System! (3)

- The whole point of understanding the system:
  - You are performing experiments with your program, and making observations of its behavior
  - You need to *correlate* the observed behaviors with various parts of your program's source code
  - The more effective you are at doing this, the faster you will zero in on the location of bugs
- (Debugging often involves a lot of thinking…)
- Both positive *and negative* correlations are helpful!
  - If you can say, "This behavior is definitely <u>not</u> caused by the code in this part of the program," that also helps narrow down the source of the issue.

Technique: Understand the System! (4)

- Memory allocators provide two major operations:
  - Allocate a chunk of memory from the heap. If requested amount of memory isn't available, simply return NULL.
  - Release a chunk of memory back to the heap.
- What would happen if I change the "release memory" code to be a no-op?
  - Allocations will succeed until all memory is consumed, and then subsequent allocations will fail
  - This is not a problem! Caller expects that allocations may fail!
- Can disable the "release memory" code, recompile and rerun the test, and see if the crash still occurs

Technique: Disabling Code

- How to disable a portion of the code depends on the specific language being used, and the program's structure
- In the C language, can comment out a chunk of code:

```
/* This is my awesome function that doesn't work. */
void foo() {
    bar();
    /* TODO: this may be buggy…
    abc();       /* Do something amazing! */
    */
    return xyz();
}
```

- C doesn't support <u>nested</u> block-comments (bit of a pain)

Technique: Disabling Code (2)

- Can also use *preprocessor directives* to disable a region of code:

```
/* This is my awesome function that doesn't work. */
void foo() {
    bar();
#if 0          /* TODO: this may be buggy… */
    abc();     /* Do something amazing! */
#endif
    return xyz();
}
```

  - The #if preprocessor directive evaluates its expression
  - If expression result is nonzero, the block of code is compiled
  - If result is zero, the block of code is excluded from compilation
  - Code within the #if 0 / #endif region will be excluded

Example: Memory Allocator

- First try: comment out the "release memory" code, and try running the allocator testing program again
- If it still crashes, what do we know?
  - We know that there's a crashing bug in the allocation code…
  - (Can't conclude that there definitely isn't a crashing bug in the deallocation code, but first things first…)
  - Can focus our attention on allocation, looking for defects
- If it doesn't crash, what do we know?
  - Again, cannot conclude that the allocation code is bug-free!
    - It may simply mangle the heap in such a way that deallocation crashes
  - Need to use another strategy to isolate the defect
- It's an easy check, and will give you more information

Technique: Create Simple Test Cases

- If you can't chop down the input data, and you can't chop down the program itself:
- Can create very simple test cases to exercise simple paths through the code
- Allocator example:
  - Write a test that allocates one chunk of memory, then frees it
    - If it crashes, you can better discern what code paths are being followed in your program
    - If not, create a more complex test case
  - Write a test that allocates two chunks of memory, then frees the first chunk
  - Repeat until you have a test case that still causes the failure

Step 3: Identify the Defect Itself

- If you have successfully completed the first two steps:
  - You probably have the location of the defect narrowed down to a relatively small portion of your code
- Now, need to identify possible origins of the failure (may be multiple candidates!), and eliminate them one by one
  - Requires that you reproduce the failure yet again, and <u>watch</u> what your program does as it goes down in flames
  - If you can't peer into your program's execution, you cannot identify the exact origins of the bug

Step 3: Identify the Defect Itself (2)

- Several different approaches for this step:
  - Instrument your code to produce logging/debugging output
  - Run your program in a debugger to single-step through the failure scenario
- Both approaches have the same fundamental goal:
  - Examine the state-changes your program is performing, correlated with the lines of code making those state-changes
  - Determine the exact point in time when your program begins to create invalid state
- Using a debugger is a more powerful and less intrusive way of doing this, but either approach will work
  - Can introduce other bugs while adding your debug output…

Technique: Printing Out Details

- A very common approach for debugging C programs:
  - Add printf() statements to the code, then compile and rerun
  - Then, pore through the debug output to see what happened
- Beware: the standard output stream (stdout) is buffered!
  - Sometimes, program seems to crash in a location not indicated by debug output!
  - Solution: flush unwritten debug output to the console

```
void buggy(int x) {
    int i;
    for (i = 0; i < x; i++) {
        printf("i = %d, a[i] = %s\n", i, a[i]);
        fflush(stdout);
        ...   /* do buggy stuff with i and a */
    }
}
```

  - (Not always necessary, but if you see odd behavior, give it a try.)

Technique: Printing Out Details (2)

- If you are going to add debug output, might as well print out everything you can think of
- Common scenario:
  - The program fails.
  - Programmer suspects a particular cause of the failure, and adds debug-output to the program to explore that specific cause.
  - Guess what, it's not that.
  - Programmer has to go back and print out more details…
- When adding debug output, print out all details that could be useful to know, so that you have full information
  - Will allow you to evaluate multiple potential causes in less time

Technique: Printing Out Details (3)

- Make sure every debug output line is unique in some way!

```
void buggy(int x) {
    int i;
    for (i = 0; i < x; i++) {
        printf("i = %d, a[i] = %s\n", i, a[i]);
        …   /* do buggy stuff with i and a */
        printf("i = %d, a[i] = %s\n", i, a[i]);
        …   /* do more buggy stuff with i and a */
    }
}
```

  - In this example, can't easily correlate debug output with the line that produced it!

- Simplest approach: put numbers or some other unique value at the front of each debug output line

```
printf("buggy 2:  i = %d, a[i] = %s\n", i, a[i]);
```

Technique: Printing Out Details (4)

- Let's say you followed these steps, and had wild success.
  - All bugs vanquished!
- And, the program now dumps out a ton of debug info. ☹
  - You need to get rid of this debug output…
    - (graders/customers <u>do not</u> want to see it!)
  - But, you may need this debug output again in the future!
- Similar to before, you should modify your debug output code so that you can <u>conditionally</u> enable and disable it
- At the top of your source file, or in a widely-used header file, define a symbol that controls debug output:

```
/* Set to 0 to disable debug output, nonzero to enable. */
#define DEBUG_INFO 1
```

Technique: Printing Out Details (5)

- Then, wrap all debug-output lines with an #if guard

```
void buggy(int x) {
    int i;
    for (i = 0; i < x; i++) {
#if DEBUG_INFO
        printf("i = %d, a[i] = %s\n", i, a[i]);
        fflush(stdout);
#endif
        ...    /* do buggy stuff with i and a */
    }
}
```

- Now, can enable/disable <u>all</u> debug output with one switch!
  - Make sure debug output is <u>disabled</u> before turning things in…

Step 4: Fix the Defect; Verify the Fix

- Once actual defect is found, usually straightforward to fix
  - However, if the program's *design* is defective, may need to rework substantial portions of the code
  - (This is why it's always good to design up front!)
- You aren't finished fixing the bug until you <u>verify</u> the fix
- By this point, you should have some way of reproducing the failure…
  - Retry your test cases and see if the failure no longer occurs
  - If no more failures, you're done!
- <u>Do not</u> assume that if the fix *compiles*, the bug is fixed!
  - If this is your approach, you actually probably made it worse…

Step 4: Fix the Defect; Verify the Fix (2)

- Bug fixes can also introduce new defects into the code
  - Such defects are called *regressions*
- Usually occurs when:
  - The actual cause of the original bug is not fully understood
  - Or, the impact of the bug-fix is not fully understood
- Good programmers also check for regressions:
  - Verify that the original bug is fixed
  - Also run other tests to ensure that no new bugs were introduced
- Be a good programmer ☺
  - It will save you <u>tons</u> of time and frustration in the long run

Some Notes about Fixing Bugs

- Programmers are an imaginative bunch.
- You must beware of some common pitfalls that are part of *your own nature!*

- **Believe your <u>observations</u>, not your suspicions.**
- A common scenario:
  - The program fails. The programmer *suspects* that the failure is caused by a particular issue, and focuses his attention there.
  - However, there is no indication that the suspected cause is the *actual* cause of the problem.
  - Sometimes this can even cause you to miss obvious details that clearly indicate the actual source of the problem.

Some Notes about Fixing Bugs (2)

- **If you didn't fix it, it isn't fixed!**
- Another common scenario:
  - An intermittent failure is occurring.
  - You have made a few stabs at fixing it, but you still really don't know what causes the problem or how to reproduce it.
  - You <u>definitely</u> can't correlate any "fixes" with the problem
- It is very appealing to assume that if a problem hasn't occurred recently, it must be fixed.
  - You may even avoid focused testing on that issue (mainly because you really don't want to know…)
- Normally, these problems come back.
  - Usually during a demo, or when your code is being graded.

Next Time: Debugging with GDB

- Have covered a relatively large range of general debugging issues and approaches
  - There is certainly much more where these came from
  - As always, practice makes perfect
- Next time, will focus on how to use GDB and Valgrind to debug your C programs