CS107 Debugging Guide

Developing good debugging practices is essential to improving your skills as a computer scientist. Every computer scientist at every level spends time debugging. **Good debugging practices will save you significant amounts of time and effort, and will make you think more systematically about your work.**

Debugging Checklist

The classic "print-statement-debugging" techniques may work for simpler cases, but quickly get unmanageable. The checklist below is an extremely effective way to find and fix bugs using debugging tools like `gdb` and `valgrind`. Debugging is a learning process - you are about to learn more about your program's behavior and how to translate your ideas into code!

View GDB Guide (/class/cs107/resources/gdb) **View Valgrind Guide (/class/cs107/resources/valgrind)**

1. **Observe the bug.** *"What makes me think there is a problem?"*
2. **Create a reproducible input.** *"How can I reliably reproduce this problem?"* You want the *smallest, simplest* test case possible that reproduces the bug. The smaller it is, the quicker you will be able to debug it.
3. **Narrow the search space.** *"How can I narrow down to the code that could cause this problem?"* Tracing through the whole program is not feasible. Instead, consider:
    - Using your intuition as a starting point. Did you change a function recently? Are there any likely culprits?
    - Using "binary search". Set a breakpoint halfway through your program. If the state is already incorrect, investigate the code before that line. Otherwise, focus your attention on the code that follows. Repeat to further narrow down.
    - Investigating memory errors. Run under Valgrind; does it point out any issues?
    - Investigating crashes. Run under GDB to let the program crash, and use commands like `backtrace` and `print` to gather information.
4. **Analyze.** *"What information can I gather from this identified code?"* Now trace through its execution with GDB. Inspect the values of variables and flow of control, and draw pictures.
5. **Devise and run experiments.** *"How can I check my hypotheses about the issue?"* Make inferences about the root cause and run experiments to validate your hypothesis. Iterate until you identify the root cause.
6. **Modify code to squash the bug**. *"How can I fix the issue, and confirm that the fix works?"* The fix should be validated by your experiments and passing the original failed test case. You should be able explain the series of facts, tests, and deductions which match the observed symptom to the root cause and the corrected code.

Do not change your code haphazardly. As a scientist, you should change only one variable at a time while experimenting. This ensures you can fully understand the program behavior and causes at every step.

Good style means easier debugging. Anything that helps you navigate and understand your code is key to efficient debugging.

Thorough testing helps uncover bugs. You can never reach step 1 if you are unaware of lurking issues!

Common Debugging Scenarios

Here are several common error scenarios and how you might apply each of the above checklist steps to fix them. For all of these, GDB is invaluable. For easier debugging, open 2 terminal windows logged into myth, with your text editor in one, and GDB in the other. Here are essential GDB commands:

Command	Description
`break [x]`	Put a breakpoint on line `x` or function `x`.
`run [args]`	Runs the current program in gdb with the specified command-line arguments.
`print [x]`	Print the value of a variable or expression.
`next`	Step to the next program line and *completely run any function(s) on that line*.
`step`	Step to the next program line, *or into the first function called by that line*.
`continue`	Continue running the program until the next breakpoint or until the program ends.
`backtrace`	Print a stack trace for where in the program you currently are.
`up` and `down`	Go up and down the stack trace to change your current view and what variables you can print. E.g. `up` places the program in the calling function, and then you can use `print` to look at the state of the program in that function.
`quit`	Quit `gdb`

My program crashes with a SIGSEGV (segmentation fault).

A segmentation fault means you are accessing memory at an address that does not belong to you.

- **Step 1:** already done :)
- **Step 2:** shrink the test case as much as possible while still preserving the crash behavior.
- **Step 3:** run the program in GDB with no breakpoints. Let it crash. Use `backtrace` to see where in your program it crashed. It may be within a library function - use `up` to go up the stack frames to where you call that function. Print out variable values relevant to the line where it crashed. Look for places you are dereferencing. Are you dereferencing an invalid address (e.g. NULL) or passing a pointer to a library function (e.g. `strlen`) that dereferences it? Identify the operation causing the crash.
- **Step 4:** if you are unsure how the operation causing the crash is occurring, set a breakpoint a few lines before the crash, run it again, and trace through execution leading up to the crash.
- **Step 5:** hypothesize how this issue is occuring. Consider trying another input to verify.
- **Step 6:** make sure you understand your fix - why the previous code was incorrect, and why this change fixes it. Try writing out an explanation to yourself! Re-run the original test case to confirm the crash is gone.

In GDB, my program crashes and shows a message with something about "unaligned", "avx", "source not found" or "no file or directory".

This means your program crashed inside a library function (which you don't have access to the source code for). This almost certainly means there was an issue with the parameters passed to that function.

- **Step 1:** already done :)
- **Step 2:** shrink the test case as much as possible while still preserving the crash behavior.
- **Step 3:** run the program in GDB with no breakpoints. Let it crash as before. When it crashes, use `backtrace` to see where in your program it crashed. The first frame(s) (e.g. #0) will likely be within a library function. Use `up` to go up the stack frames to where you call that function. Print out variable values relevant to the line where it crashed. In particular, examine the parameters you are passing to the library function on that line. Identify the parameter(s) you believe are invalid.
- **Step 4:** if you are unsure how that operation is occurring, set a breakpoint a few lines before the crash, run it again, and trace through execution leading up to the crash.
- **Step 5:** hypothesize how this issue is occuring. Consider trying another input to verify.
- **Step 6:** make sure you understand your fix - why the previous code was incorrect, and why this change fixes it. Try writing out an explanation to yourself! Re-run the original test case to confirm the crash is gone.

My program is stuck in an infinite loop.

- **Step 1:** already done :)
- **Step 2:** shrink the test case as much as possible while still preserving the stalling behavior.
- **Step 3:** run the program in GDB with no breakpoints. Let it stall. Hit Ctl-c ("control key + c") to terminate the program at that moment. GDB will then prompt you for a command. Use `backtrace` to see where in your program it was when you terminated it. Identify the loop that is never exiting.
- **Step 4:** Set a breakpoint right before that loop executes, run it again, and trace through execution leading up to the stall. Alternatively, if the stall happens only after many correct executions (e.g. the loop should loop 1000 times, but goes forever), try adding a conditional breakpoint like this: `b LINE if i == 999` . You can add a condition and the breakpoint will only trigger when that condition is true.
- **Step 5:** hypothesize how this issue is occuring. Consider trying another input to verify.
- **Step 6:** make sure you understand your fix - why the previous code was incorrect, and why this change fixes it. Try writing out an explanation to yourself! Re-run the original test case to confirm the infinite loop is gone.

My program has varying behavior (e.g. randomly inconsistent, different in- vs. out-side of sanitycheck, or in- vs. out-side of GDB) or prints weird output.

Inconsistent behavior usually indicates a memory error.

- **Step 1:** run Valgrind to see if there are any memory errors.
- **Step 2:** shrink the test case as much as possible while still preserving the memory error.
- **Step 3:** Investigate the valgrind output for the code location of the error and other helpful information.
- **Step 4:** If needed, set a breakpoint before that part of your code in GDB, and trace through execution leading up to the memory error. Try to observe the operation that causes the memory error.
- **Step 5:** hypothesize how this issue is occuring. Consider trying another input to verify.
- **Step 6:** make sure you understand your fix - why the previous code was incorrect, and why this change fixes it. Try writing out an explanation to yourself! Re-run the original test case to confirm the memory error is gone.

My program has a memory error "invalid read/write of size X"

This means you are reading to or writing from a memory location that does not belong to you. This commonly happens with heap memory (going beyond your allocated space).

The valgrind message may look something like this:

```
==3612603== Invalid write of size 1
==3612603==    at 0x483F0BE: strcpy (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3612603==    by 0x109283: make_error_1 (buggy.c:25)
==3612603==    by 0x1093E3: main (buggy.c:64)
==3612603==  Address 0x4a5f488 is 0 bytes after a block of size 8
```

Key #1: look at how large the invalid amount is (eg. 1 here). That can clue you in as to what the write operation is (eg. 1, perhaps writing a char. 8, perhaps writing a pointer or long).

Key #2: look at the trace. Valgrind tells you where in your code the read/write occurred.

Key #3: look at the address. Remember that address value and see if you see any similar addresses in your code when you step through it. That could hint at where the invalid access is happening.

See the tips for "My program has varying behavior" above for more steps.

My program has a memory error "conditional jump or move depends on uninitialized value(s)"

This means the program is executing conditionals (like `if` statements) whose outcome is dependent on uninitialized memory. A common example is calling `strlen` on a string without a null terminator, where it continues reading into uninitialized memory in search of a `\0`.

The valgrind message may look something like this:

```
==3270903== Conditional jump or move depends on uninitialised value(s)
==3270903==    at 0x483EF58: strlen (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3270903==    by 0x109263: makeError2 (buggy.c:25)
==3270903==    by 0x10930B: main (buggy.c:42)
```

Key: look at the trace. Valgrind tells you where in your code the conditional happened.

See the tips for "My program has varying behavior" above for more steps.

My program crashes with a heap error "realloc: invalid next size"

This usually means your size parameter is too large (usually happens if you `realloc` in a loop that runs way too many times) or the heap has been corrupted somewhere else and is only causing an issue now. For instance, maybe a memory error occurred elsewhere that overwrote part of the heap that is now needed to handle this reallocation request. Try investigating what other memory errors may be present.

See the tips for "My program has varying behavior" above for more steps.

See Julia Evans's **illustration on debugging (https://drawings.jvns.ca/better-debugging/)**.