General Debugging Practices in C

1 Intro

Welcome to the wild world of C! Fortunately, all of the debugging tips for C0 you learned in the Guide to Debugging C0 Code are still applicable in C. As such, this guide will focus on certain common C-only errors and provide tips on how to debug your code when you encounter one of these errors. Of course, it is impossible for such a guide to go over every possible error!

2 A Segmentation Fault has Occurred

If a segmentation fault occurs, this is fundamentally because somewhere memory was accessed incorrectly. Some of the most common causes for this are:

- Dereferencing a NULL pointer
- Dereferencing a pointer whose value is undefined
- Attempting to write to read-only memory
- Out of bounds array accesses

If you get a segmentation fault, your first step should be to isolate exactly what line caused the segmentation fault, and what about that line caused the segmentation fault. If you have a suspicion about what the error is, it could be helpful to add print statements near the lines you think might be causing the error to verify exactly which line is the problem. See the Printing in C Guide for the mechanics of printing in C, and the Debugging with Print Statements Guide for helpful tips on where to add print statements (how you print is a bit different in C but the tips about how to use print statements remain the same). If you have no idea where your error comes from, then using Valgrind can help you find the error as well as pinpoint exactly what kind of error it is. For help on how to use Valgrind, see the How to Use Valgrind guide.

Once you have determined the line and what about the line caused the bug, the techniques discussed in the C0 Debugging guide (Guide to Debugging C0 Code) will once again aid you in resolving the bug.

3 My code does not do the same thing every time I run it

If you are not recompiling your code (or recompiling without changing it), and yet the result changes unexpectedly, you are most likely encountering undefined behavior! If this is happening, using Valgrind can be a great way to identify where your undefined behaviour is happening. For help on how to use Valgrind, see the How to Use Valgrind guide.

One common reason for this to happen would be if you use an uninitialized variable as if it were initialized. As an example, the following code will sometimes print `"Hello!"`, and sometimes `"Goodbye!"`:

```c
int x;
if(x == 0) {
    printf("Hello!\n");
} else {
    printf("Goodbye!\n");
}
```

4 Big Scary Memory Dump

When we say "big scary memory dump", we mean something that looks like this:

```
*** Error in ‘./a.out’: free(): invalid pointer: 0x0000000000400680 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x81299)[0x7fccb0755299]
./a.out[0x400599]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7fccb06f6555]
./a.out[0x4004b9]
======= Memory map: ========
00400000-00401000 r-xp 00000000 00:28 1992317346                         /afs/andrew.cmu.edu
00600000-00601000 r--p 00000000 00:28 1992317346                         /afs/andrew.cmu.edu
00601000-00602000 rw-p 00001000 00:28 1992317346                         /afs/andrew.cmu.edu
7fccac000000-7fccac021000 rw-p 00000000 00:00 0
7fccac021000-7fccb0000000 ---p 00000000 00:00 0
7fccb04be000-7fccb04d3000 r-xp 00000000 fd:00 12073754                   /usr/lib64/libgcc_s
7fccb04d3000-7fccb06d2000 ---p 00015000 fd:00 12073754                   /usr/lib64/libgcc_s
7fccb06d2000-7fccb06d3000 r--p 00014000 fd:00 12073754                   /usr/lib64/libgcc_s
7fccb06d3000-7fccb06d4000 rw-p 00015000 fd:00 12073754                   /usr/lib64/libgcc_s
7fccb06d4000-7fccb0897000 r-xp 00000000 fd:00 12059270                   /usr/lib64/libc-2.1
7fccb0897000-7fccb0a97000 ---p 001c3000 fd:00 12059270                   /usr/lib64/libc-2.1
7fccb0a97000-7fccb0a9b000 r--p 001c3000 fd:00 12059270                   /usr/lib64/libc-2.1
7fccb0a9b000-7fccb0a9d000 rw-p 001c7000 fd:00 12059270                   /usr/lib64/libc-2.1
7fccb0a9d000-7fccb0aa2000 rw-p 00000000 00:00 0
7fccb0aa2000-7fccb0ac4000 r-xp 00000000 fd:00 12059263                   /usr/lib64/ld-2.17
7fccb0c88000-7fccb0c8b000 rw-p 00000000 00:00 0
7fccb0cc1000-7fccb0cc3000 rw-p 00000000 00:00 0
7fccb0cc3000-7fccb0cc4000 r--p 00021000 fd:00 12059263                   /usr/lib64/ld-2.17
7fccb0cc4000-7fccb0cc5000 rw-p 00022000 fd:00 12059263                   /usr/lib64/ld-2.17
7fccb0cc5000-7fccb0cc6000 rw-p 00000000 00:00 0
7ffdcf716000-7ffdcf738000 rw-p 00000000 00:00 0                          [stack]
7ffdcf77a000-7ffdcf77c000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

If this is happening to you, the most likely reason is an error related to malloc/free like freeing something twice or freeing something that shouldn’t be freed. The way to get a more understandable error is to re-run the program with valgrind. This will provide a more clear error with useful debugging information. For help on how to use Valgrind, see the How to Use Valgrind guide.

5 Integers suddenly having unexpected values

If you are working with integers and you notice that suddenly your integers have extremely unexpected values[1], then you most likely encountered a casting issue. Make sure that:

- You are always casting explicitly
- When you perform arithmetic operations, the two integers being operated on have the same type.
- You are using an appropriate integer type for what your integer is supposed to represent.

In making sure that you are casting correctly, using print statements can help elucidate exactly what is going on - just make sure to match the length sub-specifier and the specifier character to the exact integer type you are using! The quick reference table in the printing in C guide can be very helpful. A direct link to the table is here.

6 General Tips and Tricks

- Make sure to *always* cast explicitly with integers!
- Before anything else, make sure you run Valgrind and don't have any errors. There are cases in which memory errors lead to other seemingly unrelated problems.
- Good data structure invariants are more important than ever - with the myriad of new possible memory errors, it is very useful to know that your data structures are correct.
- In general, make sure you don't forget to use contracts!
- The `lib/contracts.h` library contains a useful macro called `IF_DEBUG`. Essentially any code you write inside the `IF_DEBUG` will only be executed when contracts are enabled. This can be useful to add prints in such a way that they do not slow down your code or clutter the output all the time. An example way to use this is below:

```
IF_DEBUG(int count = 0); // Count is only defined when contracts are enabled
while(!stack_empty(S)) {
    IF_DEBUG(printf("Inside loop for %d times!\n", count)); // Only printed when contra

    // Some code dealing with the stack

    IF_DEBUG(count++); // Count is only incremented when contracts are enabled
}
```

- Sometimes, the value of a pointer can be a hint as to what type of memory you are dealing with:
    - If the value starts with `0x400` (e.g. `0x400650`) then the pointer likely points to read-only memory (such as for string literals

[1] For instance, if you are working with a variable that has been steadily increasing by quantities smaller than 5 and all of a sudden it increments by 2 billion, then that would count as an unexpected value

- If the value is between `0x1000000` and `0x10000000` (e.g. `0x16e0010`), then the pointer likely points to a location in the heap. In other words, the pointer is likely one returned by malloc
- If the value starts with `0x7ff` and is very long (e.g. `0x7ffd592511f0`) then it is likely a stack address
- If the value of a pointer is anything else, then it is likely a garbage value.