From CS61Wiki

Contents

Learning Objectives

- Identify what program behavior is governed by C and what is governed by lower layers of abstraction.
- Use gdb to examine memory.
- Explain how and why optimized and unoptimized code can behave differently in GDB.
- Explain the representations underlying different abstractions in a C program.

A Meta Comment

Please don't peak ahead.

You should work on these exercises in groups of 2 or 3.

In-class exercises are not graded. It would be possible for you to "cheat" on them -- e.g., run programs when we ask you to read them, read ahead when we might answer questions that we ask you to think about. This will not help you do well in the course -- it's way too easy to look at code after you know what it does and say, "Oh yeah, I knew that," when, in fact, sometimes programs can be tricky and deceptive. So please, complete the exercises as we ask you to.

If you get stuck, if there are constructs in the C code you don't understand, if there is a command we ask you to use and after reading the man page, you don't understand it, or if there is disagreement in your group, raise your hand and a member of the teaching staff will come by to help get you unstuck. The goal is that you understand what's happening, and we (the staff) are here to help you do that.

Exercise 1

For each main program below (each one is numbered so you and we can talk about them more easily), predict what will happen were you to compile and run the program.

```
int main0(void) {
```

```
   char ch = 'A';
   fprintf(stderr, "%c\n", ch);
```

A - predictable

```
}

int main1(void) {
```

```
   char ch = 'A';
   char* ptr = &ch;
   fprintf(stderr, "%c\n", *ptr);
```

A - predictable

```
}

int main2(void) {
```

```
   char ch = 'A';
   char* ptr = &ch;
   /* %p is the way you print a pointer value in hexadecimal */
   fprintf(stderr, "%p\n", ptr);
```

The address of some variable on the stack, but not specified in language.

```
}

int main3(void) {
```

```
   char ch = 'A';
   char* ptr = &ch;
   printf("%c\n", ptr[8]);
```

Really bad. That's not even a location we should access!

```
}

int main4(void) {
```

```
   char ch = 'A';
   char* ptr = &ch;
   ptr[8] = 'B';
   printf("%c\n", ptr[8]);
```

Oh no... now we're scribbling on random memory.

```
}
```

Questions

1. If there were any programs for which you could not predict program behavior, explain why.

2. What does your inability to predict behavior tell you about the difference between the abstractions in C and the real machine?

We can only predict things specified in the C language - things at lower layers of abstraction are not predictable

Important: Some of the programs did things that are *undefined* in the C language. **When the compiler sees undefined behavior, it is allowed to do anything it wants.** In an ideal world, the compiler will tell you when your programs exhibit undefined behavior -- in many cases, these will appear as warnings. It is in your best interest to fix your code so that it does not generate any warnings when you compile it. However, sometimes you will write code exhibiting undefined behavior and the compiler may not realize it -- this is when an understanding of the real machine and gdb will be quite valuable.

Running code and using GDB

The rest of these exercises should be completed on the appliance. If you have not completed the pre-class work, you will need to follow the instructions on the Infrastructure page to continue.

Now, clone the cs61-exercise repository from code.seas

```
git clone git@code.seas.harvard.edu:cs61/cs61-exercises.git
```

Go into the `l02` directory.

Build the five programs using `make` and run them. Were your predictions correct? Do the programs produce the same output each time?

Let's now look at the program `main1` in `gdb`. Run `gdb` on the program, set a breakpoint at the `fprintf`, run the program and print out the value of `ptr`.

```
gdb main1
Reading symbols from main1...done
(gdb) b 7
run
Breakpoint 1 at 0x8048440: file main1.c, line 7.
(gdb) c
```

The compiler optimized out the variable - you don't really need it.

What happened? With your group, devise an explanation for what happened and why.

Compiler Optimizations

The `ptr` variable isn't strictly necessary in `main1.c` -- it references another variable whose value and the compiler knows that the contents of pointer are exactly the same as the contents of the ch variable, so it doesn't even bother allocating space for the ptr. Let's tell the compiler not to optimize our code and see what happens.

Edit the Makefile and change the "`-O3`" in the `CFLAGS` declaration to "`-O0`". The `-O` flag specifies an optimization level -- lower numbers are less optimization; 0 means do not optimize at all. Now, run `make clean` to remove the prior versions of your programs and then run make to rebuild them all without optimization. Now, run main1 in gdb and see what happens when you try to print out the value of `ptr`.

Hint: If you are debugging a program and you find that the debugger isn't showing you what you want, recompile with `-O0`.

hexdump: a handy function

You will notice a file in the `l02` directory named `hex dump`.

I used the hex dump function in the abstraction video, but you might not have internalized exactly what it's showing you, so let's take a look at that now. The program `main5.c` is identical to the one from the video, but let's take a minute to really understand the output. Running `main5` produces something like this (the address of the local and heap variables may be a bit different as you recall):

```
bfde9e77 41 |A|
0804a024 42 |B|
08048870 43 |C|
0830e008 44 |D|
```

The first column is the address -- the location in memory (the process's virtual address space) that we are displaying. The second column prints the value stored at that address in hexadecimal. The last column prints the value in ascii. (On most systems, you can type `man ascii` and see the mapping between various number bases and the ascii characters; those man pages do not appear to be installed on your appliance though.) Unprintable characters appear as .

Next, let's use `hexdump` to figure out why a program crashed. `main6` is identical to `main4`, except that it uses `hexdump` to examine memory. Look at the difference between the two displays; can you see where the value 'B' was written? Take some time to answer the questions below -- when have answers to all of them, raise your hand and check in with a member of the teaching staff.

Questions

This value will change each time you run - the key is that it is 8 bytes after the start of the array

1. What is the address that was overwritten with the value 'B'? 0xbfb9f8f8 = 7 + 8
2. What value was in that location before it was overwritten? b7
3. Let's assume that that location is the first byte of a 4-byte quantity -- what is the 4-byte quantity stored starting at that location? 0xb754da83
4. In which segment (part) of the address space do you think that value resides? STACK
5. What do you think that value represents? RETURN ADDRESS
6. Why did your program crash? We returned to a bogus address.

You may use `gdb` to test your hypotheses. Hints: `bt` will show you the sequence of function calls that have been made to get you to the current point. `s` will let you single step through your program; `n` is just like `s` except it treats a function call as a single step, while `s` will step into the function. If you inadvertently go into a function that you didn't intend to, `finish` will get you back to the caller.

Checkpoint

At this point, you should have accomplished the following:

- You can identify some undefined behavior in C and what kinds of things can happen when your code expresses those undefined behaviors.
- You can use `hexdump` to examine the contents of memory.

- You have a sense of how an address space is laid out and can make good, educated guesses about the segment in which an address exists.
- You can use gdb to examine memory and trace program behavior to understand what's happening.
- You understand that the compiler can optimize code and that these optimizations can sometimes make debugging more difficult than it should be; you know how to disable these optimizations.

If you aren't feeling confident about these things, please raise your hand and ask a staff member for assistance.

Exercise 2

Now, we want you to experiment with the `main6.c` program, making modifications and seeing how its behavior changes.

Try each of the following experiments by modifying main6.c, rebuilding it, running it, and potentially examining it in GDB.

1. Change the 'B' in line 10 to a 'C'. What happens? How did the program behavior change? (Hint: It did change, so if you think it didn't, take a peek inside gdb.) Try some other values and see what happens. Some interesting values to try: 0xff, 0, 0x08. It fails at some other address.

2. Here is a particularly fun one: Replace line 10 with the following lines:

```
ptr[8] = 0x08;
ptr[7] = 0x04;
ptr[6] = 0x84;
ptr[5] = 0x70;
```

It went back to main'. We put the address of main on the stack as the return value.

What happened? Why?

3. So far we've used hexdump primarily to examine the memory around local variables; let's examine memory around some other things. `main7.c` is quite similar to `main5.c` (the program from the video), but instead of just printing out a single character; let's print out a chunk of memory. Build and run the program.

Notice that you can still see the characters that we saw when we ran `main5`, but we see that the memory locations after then also contain "stuff." All memory that a program can access contains "stuff" -- it may just not be stuff you care about. Notice the last `hexdump` -- we asked it to dump out memory at main. What do you suppose that "stuff" represents?

4. Let's add a few more variables in our program and see what happens. Take a look at `main8.c`. Build (you may ignore the warnings) and run it. Can you find the new values that we added? If you don't see all the values, edit `main8.c` to print out more data and see if you can find the missing values!

5. By now you know that the constant globals and non-constant globals show up in different places in the address space (they are relatively close to each other, but not right next to each other). Suppose you try to change the value of a constant global, what do you suppose will happen? Make a prediction before editing `main8.c` to try this out. Compiler catches this

6. Finally, let's see what happens if we try to display memory that doesn't seem to correspond to any of our variables or code segments. What happens? Write a small test program that:

- Tries to print out the memory starting at 0xdeadbeef.
- Tries to print out the memory at an address between your heap and your stack (have your program compute this; do not hardcode it in).
- Tries to print out as much of the stack as you can find.

What happens in each case? Many of these are bad addresses & we crash

Some Take Aways

- Programs execute as processes running in an address space.
- An address space is just a way to describe memory.
- Memory just contains numbers; it is the abstractions in C (or any language) that give those numbers meaning.
- Some numbers in memory correspond to addresses; some correspond to instructions; some correspond to values you've set in your program; others seem to appear out of nowhere.
- Some parts of the address space are inaccessible -- programs crash trying to read memory there.

Before leaving class, please take 30 seconds to complete this survey (https://docs.google.com/a/g.harvard.edu/forms/d/1Wsj0D7RnDoQbLTvYUI5O5ePjml60Xlw3bTUrBuKad-Y/viewform) .