CS 61 Lecture 4 Scribe Notes

Eddie Kohler
Scribe: Duligur Ibeling

N.B. Code examples (e.g. the complete `factorial` *function) can be found on the main CS 61 website.*

1 Administrative announcements

A good number of people are now up to test 6. Check-in will be tomorrow.

Scribe notes are due a week from the class scribed.

2 Types and pointer arithmetic

When I add one to a pointer, the result is the address plus the number of bytes in the object pointed to.

But when I get a pointer from malloc, it doesn't have a type. What happens then? Where does the truth lie? There is a standard, which in fact claims that you cannot do arithmetic with a `void *` pointer. You must cast it to a different type.

What's the best type? It's `char *`. Instead of doing

```
void * x = malloc(sz);
```

after which you might try to do `x + 1`, which would call up the wrath of the nasal demons, you should try something like

```
char * x = (char *) malloc(sz);
```

`x+1` just adds 1 to the address now. The size of a `char` is 1.

`unsigned x = (unsigned) malloc(sz);` doesn't work on all machines. `unsigned` is the same size on all machines we'll use, but `void *` is 4 bytes on a 32-bit (VM) and 8 bytes on the 64-bit (raw machine). Casting `void *` $\rightarrow$ `unsigned` throws away half the bits in the address. Dangerous.

Address arithmetic with numbers should use `uintptr_t` type. See the C patterns page.

Let's do an experiment. Let's call malloc in 4 ways:

```
                    void *x = malloc(1);
                    char *y = malloc(2);
          char *z = (char *) malloc(3);
unsigned gibberish = malloc(4);
```

What happens? Make it and include all warnings. Initially we get unused variable warnings. Cast them to void by adding

```
(void) x, (void) y, (void) z, (void) gibberish;
```

This is actually stylistic. Says it's ok that these aren't used.

But now we get a warning that an initialization makes integer from pointer without a cast. The only place in which the cast is strictly necessary is with `gibberish`, when we take a pointer and turn it into an integer. But try to cast the return type of `malloc` to the type it needs, for stylistic reasons.

3 Clarifications on `memset`

Third argument of `memset` is the number of bytes to which a certain amount of data will be written. But what if you write data into fewer bytes than the data itself takes? What does the third argument really do?

We already answered our own question, but let's try crashing a program. Declare `int z;` and `memset` it as

```
memset(&z, 61, sizeof(int));
```

Let's recall that `sizeof` gives the number of bytes required to hold a type or something of that type. So `sizeof(z)` would be fine here too.

Let's print using a standard `printf("%d\n", z);`. We get junk. Not `61`. We got `3D3D3D3D`, which is `61` repeatedly in hexadecimal. We got just what was expected (copies of 61). What else could `memset` do?

If a variable is cast to `void`, can it still be used later? Yes. It doesn't kill the variable. It's a use, for the compiler and human readers, but the use doesn't do anything. It just makes the compiler shut up.

We saw a good use of `memset`. Another use? Try `memset(&z, 61, 61);`. We get the same printout, and then it segfaults. We tried to access memory outside of what we said was for `z`.

Why exactly does `memset` fill with copies of `61`? `z` is on the stack. Somewhere below `0xC0000000`, we have `z`.

What is `memset`? Go `man memset`. `man` gives info about pretty much all function or system calls. Could also Google, but `man`'s better. It doesn't require the internet.

The manual specifies that `memset` fills the area with the bytes. Thus, first, we put `61` into all the bytes of `z`. We get a big number! (`61` in all bytes of `z`.) But since the third argument was `61`, we keep writing `61`, going into portions that don't correspond to `z`. Undefined behavior! Don't want this. We could get anything.

But why did we print first and not crash immediately? Undefined behavior is triggered, like setting a bomb's fuse. The bomb can go off anytime. In our case, once `main` tried to return, the return information was overwritten with `61` and the program crashed.

We could cause the undefined behavior to happen earlier. Just write a lot more bytes of `61` by making the third argument of `memset` some insane number. Printout no longer occurs, we just segfault and core dump. Why did we crash earlier? It ran into the kernel's memory space. That is a very bad thing. The OS prevents us from doing it right away.

Objects in memory are laid out from left to right (heap or local or global or constant global), but we need to distinguish between objects in memory and stack frames. The former must be the case since we must be able to take pointers. However we use the code, it must be invariant with regards to where the pointer points (as long as the pointer points somewhere).

Clarification: we can make bytes `61` since $0 \leq 61 \leq 255$. If we do something that won't fit in a byte for the second argument, `memset` will truncate it. Example: fill with `61616161`. This is `3AC3021` in hex. `memset` filled with `33` in decimal, which is `21` in hex. I.e. the last two digits of the above. So we can see that truncation occurred.

Note: bin $\rightarrow$ hex conversion can be done with `dc` on the command line.

This is the wrong way to do this with `memset`. Is there a right way? What were we trying to do? If it was just setting an integer to some value, keep it simple and do `z = 0;`. Sanity check: we get `0` as the output.

In the default `m61.c`, we set statistics to garbage using `memset` to insure that it doesn't automatically work, and we have to do something.

Note that `memset` writes `61` as a `char` and not an `int` because this is just how `memset` works. If we go back to the manual page, it says it fills bytes with the constant byte `c`, where `c` is the second argument.

Looking these up in the manual helps a lot.

4 Stack frames

4.1 An illustrative example

Let's look at a naive factorial function `int factorial(int n)`. We know that $0! = 1$, so in `factorial(n)`, check if `n` is 0, and if it is, return 1. Otherwise, we have the (recursive) relation $n! = n(n-1)!$. Thus we can write the whole function as

```
int factorial(int n) {
    int x;
    if (n == 0)
        x = 1;
    else
        x = factorial(n-1) * n;
    return x;
}
```

We can go up to about 12!. 13! is wrong. 14! is smaller than 13!. Wrong results, something deeply finite is happening.

But we are using factorial to investigate the stack discipline (both its benefits and costs). Also, last lecture, there were a couple examples that we didn't quite do right, and we'll do them right this time.

Local variables in a function are stored in the stack. All of the local varibles for some active execution of a function are in the stack frame. Think of this as a box. I.e. a stack frame is a contiguous region of stack memory holding local varibles for an active (in the process of executing) function.

If we run `factorial(3);`, how many stack frames (maximum) do I have? The answer is 4, for $n = 3, 2, 1, 0$.

How do we figure out how big the stack frames are? We could try printing the address of the function each time, `printf("%p\n", &factorial);`. We get the same address each time. Well, we printed the address of the code, which is invariant. We need to print the address of a local variable in the code. Let's choose `printf("%p\n", &n);` in the `factorial` function. Now we get varying addresses, `0xbf82a9e0, 0xbf82a9b0, 0xbf82a980, 0xbf82a950` specifically. The stack frame's size is 48. Subtract the last two numbers to get `30` in hex, which is 48. Can quickly check the previous pairs give the same difference.

Calling `factorial(3);` makes a stack frame. It calls `factorial(2);` which makes another stack frame at a lower address (stack grows *downwards*, heap grows upwards). Somewhere in the first stack frame, we have a `3` corresponding to `n`. Somewhere in the second stack frame, we have a `2`. Of course, we have `factorial(1);` and `factorial(0);` stack frames, at lower addresses from the ones before. We go: call, call, call, return, return, return. Each call moves the stack frame to a lower address, and each return moves the stack frame to a higher address. That is:

$$\xleftarrow{\text{CALL}} \qquad \xrightarrow{\text{RETURN}}$$

4.2 The debugger `gdb`

Let's try using `gdb` to find the same information. Run `gdb factorial` (assuming the name of the program is `factorial`). Then type `r 3` in `gdb` to run the program with argument $n = 3$.

But `gdb` gives no information. We need to run the program gradually and examine the contents of memory. We need to set a breakpoint, say at the function factorial: `b factorial` in `gdb`. Then `r 3` again. We get a breakpoint with $n = 3$. We can print the value of n with `p n`. (`x/i n` for examine the value of n as an integer doesn't work, let's return to this later.)

We wanted to print the address of n, so we do `p &n`. We need to now step through the code gradually. Type `s` repeatedly in `gdb` to step through. It steps through calling factorial and hits the breakpoint, calling factorial again with $n = 2$. Doing `p &n` again, the value of `n` is different, since it's on a different stack frame.

How do we see the number of stack frames? We can use backtrace. `printf` debugging is really useful, but this is really useful too. Do `bt` in `gdb`. It shows that we came to `factorial(3);` from `main`. We can print the value of n in one stack frame by doing `p &n`, and then go up one level with `up`. This moves to the context of the previous function. Doing `p &n` gives us another value.

Now keep typing `c` for continue to go down the rabbit hole further. Eventually we hit the $n = 0$ call. Then do a `bt`. We see the addresses

`0x0804846f` ($n = 1$ `factorial`),
`0x0804846f` ($n = 2$ `factorial`),
`0x0804846f` ($n = 3$ `factorial`),
`0x080484c2` (`main`).

4.3 Implementation of stack frames

Why are the first three above the same address? What is the compiler's job? Well, one of them is to create space for every object. Needs to allocate space, space can't overlap with other objects. For globals, it's easy to see how this happens. But for function local variables, how does the function find *its* value of n? We need 4 different ones in this case, and they must be at different addresses. How is the compiler solving this problem? I.e. how are local varibles for different executions of a function distinguished? We have the same address three times above, but the local variables are clearly different.

Recall that *any problem in computer science can be solved with another layer of indirection.* We could have a pointer after every stack frame pointing to where the current stack frame is. Or a global variable pointing to the current stack frame. When we call a function, we shift this global variable down. When we return from a function, we shift it up (see the above discussion on calling/returning). If all the access through local variables is through this global variable pointer, then we have another layer of indirection!

Suppose the stack frame looks something like (in pseudo-C)

```
struct stackframe {
    int n;
```

```
    int x;
    struct stackframe * caller;
}
```

The meanings of `n` and `x` are just those in the `factorial` function. The meaning of `caller` is that it points to the function that called the function, this stackframe (execution).

Let's add a global

```
struct stackframe * esp;
```

If now the compiler says `esp -> n` (or the equivalent of this in machine code, of course), then we get the proper value of n for the current stack frame.

This idea is so important that the global variable `esp` is induced by the architecture. The processor has the global variable built in. It is on register memory, the fastest and most expensive kind of memory.

Let's take a look at the registers. Go `info registers` in `gdb`. We get output

```
esp         0xbffff170        0xbffff170
```

There are only about 20 registers on our machines, and they are effectively global. Only about 6 can be used. The global variable we're talking about is the stack pointer. It's used as a base to refer to different local variables.

We can't actually write to the register variables in C, they're owned by the compiler. We could do it in assembly, though.

Let's run the program again. Do `d 1` to delete our old breakpoint. Set a breakpoint at main: `b main`. Run the program with `r` and then `info registers`. We get

```
esp         0xbffff230        0xbffff230
```

and doing `p &n` gives us `0xbffff24c`, which is above the stack pointer. The stack pointer points to the bottom (smallest address) in the current stack frame, and local variables have greater (higher) addresses.

Step a bit more with `n` (next). Step into `factorial`, and do `info registers` again:

```
esp         0xbffff200        0xbffff200
```

Note that the addresses are different (in fact, this latter one is *below* the former one, consistent with our "calls go downwards" dogma), and again doing `p &n` produces `0xbffff230`, which is in the stack frame pointed to by `esp`.

4.4 Compiler optimization

Why is the stack frame 48 bytes if we've only got 2 variables? We're not optimizing, we had the compiler compile dumbly. The version of factorial compiled with optimization is extremely confusing. Setting a breakpoint at

the `factorial` function and running without an argument doesn't even break. Running `r 10` and stepping with `c` doesn't break multiple times! This is odd. What happened is that the `gcc` people are highly optimizing the code coming from the compiler. They are trying to output code that is as fast as possible.

So the fastest way to run the code is not a direct translation of the code! But this is fine. The job of the compiler is not to translate the code literally, but to produce code that *has the same effects*. Stack frame things are below the level of the abstract machine. If we imagine the compiler's code separately from a direct translation of our code, and they *have the same effects*, then we can't tell the difference between the two (from the outside, i.e. at the level of the abstract machine) and the compiler is doing its job. And giving us a speed boost, too.

In our case, the compiler has actually taken the factorial function and turned it into a loop. This is possible because we can transform the factorial to be tail recursive (doesn't require extra work after it returns). We can add some kind of accumulator variable, as so:

```
int factorial(int n, int current_answer) {
    int x;
    if (n == 0)
        return current_answer * 1;
    else
        return factorial(n-1, n * current_answer);
}
```

Now in every branch nothing is done after the recursive call (we don't need to multiply by n after the recursive call, as we did in our original `factorial`). The compiler also took out 2 as a special case, since we set n to 2 by default. So the compiler has hard coded the result when $n = 2$ so that it doesn't even have to call `factorial` when n is 2.

Often, the optimized code will be bigger. So we are doing some kind of time-space tradeoff here. But there is a specific flag we can use to specify that we want the smallest code possible. But in modern code, often these optimizations actually shrink the code.

An interesting question is whether it is always possible to convert any recursive function into a non-recursive form, as we have done with `factorial` above. The answer turns out to be yes. Well, trivially, we can just implement our own stack data structure to emulate the call stack (we can often do better than this, though). More interestingly we could look at this in terms of the theory of computability. We would show that recursion and some reasonable non-recursive system are both *Turing complete*, meaning they are equivalent in the set of functions they can compute. So the answer must be yes, although this probably tells us nothing whatever about how to do the conversion *intelligently*.

Taking compilers will teach you a lot more about the cool subject of optimization!

5 Data representation

Data representation is not the where question (we discussed a where question above). Data representation is the how question: *how are values represented in memory?*

5.1 A simple example

Let's start with a simple example: `char x = 61;` How is this represented? The answer's simple. `char` is a byte (part of the C abstract machine), and a byte is 8 bits. So we just have a byte with `61` in it.

Let's use the program we just wrote called `datarep`. It investigates data representations, hexdumping some specific area. `hd` is one of the very useful programs. It does a hexdump. It gives addresses (offsets) in the left column, hex values in the middle columns, and characters (if they're real) in the right column. `hd` and `datarep` do similar things.

Running our program gives us the character `a` as we should expect, which has a hex value `61` (this is pretty consistent across encodings). Let's now set `int x = 'a';` in `datarep.c` and try to hexdump this.

An int is 4 bytes in memory, because the maximum value of an int is 2^{31}. (One bit is reserved because we also need to store the sign of the int.) In `int x = 'a';`, the `'a'` is hex `61`. Where does the `61` go? We have 4 bytes to an int. It would not be good to try to put into one of the middle bytes, so does it go in the first or last byte?

Well, we could do it either way. The names of these ways are *big endian* (more significant digits are on the left, which is the way we write numbers) or *little endian* (more significant digits on the right). x86 is little endian. Running our new version of `datarep` confirms this.

There was a historical war about little vs. big endian. In the end, both were used. Intel picked little endian, and the internet picked big endian. Be aware that both are used, but for the rest of the class we'll mostly use little endian. The names actually come from *Gulliver's Travels*, where they describe the separate parties in a quarrel over whether one should break an egg on the larger or smaller end. `;)`

5.2 Types of integers

In C, what types of integers are there? A lot. `short`s, `long`s, etc. A `char` is one byte, a `short` is 2 bytes, an `int` is 4 bytes, a `long` (on 32-bit machines) is 4 bytes, and a `long long` is 8 bytes.

We can examine the `long long` with `61` in it with `datarep`. We confirm that it's little endian.

5.3 Pointer representation

Let's use `datarep` to look at pointers. In `datarep.c`, do something like `int y = 'a'; int * x = &y;` and `hexdump(stdout, &x, sizeof(x));`.

We expect 4 bytes, because in a 32-bit architecture memory addresses are 4 bytes.

Running the new `datarep` produces

```
bf83eadc        d8 ea 83 bf
```

in the first two columns. These are almost the same. What is the difference? It's 4 bytes, because the thing points to an `int`. We first put `y` down, then 4 bytes later, we write down `x` which points to `y`.

Let's change `y` with `int y[1999] = {0};` instead and make `int * x = &y[0];`. What could be the differences between the addresses of these local variables? It depends on whether `x` or `y` comes first. So it could be 4 or about 8000 (if the array instead comes first). It's 4. The compiler is allocating things from the bottom up. But we could get the other result.

So pointers just stored as addresses.

5.4 Arrays

How are arrays represented? Let's do `int x[2] = {0, 1};` in `datarep.c` and get the hexdump at `&x`. We see

```
bf837a38        00 00 00 00 01 00 00 00
```

as we should expect. 4 bytes that are 0, and 4 bytes that are 1 (remember that this is little endian).

They're right next to each other. Arrays are *contiguous* in memory. This is crucial. It's what makes arrays nice. They are nice to the cache, and we can access more of the array at once with the cache. Contiguous memory is easier to optimize.

Let's look at another array,

```
char x[] = "Hello, this is a string!";
```

Running `datarep`, we can see all the characters are just laid out next to each other in memory.