CS61 Lecture II: Data Representation
with Ruth Fong, Stephen Turban and Evan Gastman

Abstract Machines vs. **Real Machines**

Abstract machine refers to the meaning of a program in a high level language (i.e. C)
Real machine refers to the meaning of a program executed on real hardware (laptop, phone, car, etc.)

eg.

Real model:

(1) You want code a program that prints the letter 'A'
(2) You code the program in c and save it as a .c file.
(3) Your compiler processes the .c file and turns it into machine instructions.
(4) Those machine instructions are taken by your processor, which directs your screen, disk, memory, CPU, etc.
(5) You have printed the letter 'A' onto your screen.

Abstract model consists only of steps (1), (2), and (5).

Why do we care about steps 3 and 4?
If we know the whole route, we can make our code become faster, more robust and more powerful and prevent against the dangerous lapses that can happen in our code.

I. Memory

- Eddie's metaphor: Memory is like an enormous post office with a bunch of boxes where each box can only hold a number between 0 and 255.
  To first order, we treat memory as **random access**. What does that mean? You can look in one box and then immediately look at another box as look as we know it's address. That is, we can jump from box to box in a unit of time (*we don't have to iterate through boxes sequentially*).
  -Address access happens in 1 unit of time. How are addresses written? Numbers in the interval [0 and 2^32).

Exercise:
Open up mexplore.c

Add a line: char* ptr = &ch;

Change following line: fprintf(stderr, "%c\n", ***ptr**);

What's this doing compared to the original function?
Answer #1: The same thing.
Answer #2: **Memory is not handled the same way.**

A smart compiler can make the pointer disappear, thus **optimizing** your program. ‘Optimized’ meant that the compiler recognizes meaning of the program (print ‘A’).

Let's use GDB to see what's going on.

GDB is a program that hooks up to a different program and allows us to run it but also pause it and snoop around in its execution.

gdb ./mexplore
Make sure you run "make" first
b mexplore:c:7
Set a breakpoint in mexplore.c at line 7

What happened?
GDB changed the program in order to allow us to stop and look at the state of the real machine
r
run
p ptr
print the value of ptr
p *ptr = 'B'
set the variable

c
continue
prints B

Now, let’s use an optimized program.

emacs -mw mexplore.c

How to create the optimized version?
make clean
make mexplore
See "-O0"? That means don't optimize
make mexplore.optimized
"-O3": optimized!
Change program again to just print ptr, not *ptr.

The optimized version doesn’t print the pointer. The compiler understands the meaning of the program and so we don’t need the pointer.

C lets us bounce between the abstract machine and the real machine (that's how we can print the address of the pointer). In OCaml, the abstract machine has very strong boundaries.

The value of the pointer keeps changing when we run the program over and over again.

Break

Current code:

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

Real machine? Print out the character that's 8 spaces away from the pointer's address.
Abstract machine? Undefined behavior.

Nasal Demons: If the compiler can prove that your program has undefined behavior, then they can make demons fly out of your nose.

Why did the segmentation fault core dump when we run ./mexplore.c?

> Writing to memory that I haven't been allowed to use on the stack (student answer)
>
> Eddie: How do we figure out what's explicitly going on? GDB!

"0x42… in ?? ()"

Segmentation Fault! *Eddie flails to mimic what happens when you run GDB.*

bt
Take a look at the backtrace.

How did we end up executing code at an address that doesn't exist?

b main

Run the code again and step through each line (using GDB)

next

r

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

Things look okay as we go line-by-line, we're about to exit main… after we return from main, we execute some weird code somewhere.

Return address

What if we overwrite the return address to contain garbage? It goes into garbage land (this is all happening in the real machine; in the abstract machine, the meaning can be anything at all)

To come in future lectures: return value bugs.

Second Example:
Change the execution of the program (Change “b” to “c”)
The segmentation fault occurs in a different address. (42 changes to 43 when we change “b” to “c”; the address printed out by GDB)

```
fprintf(stderr, “%c\n”, ptr[8]); //c stands for “character
fprintf(stderr, “%d\n”, ptr[8]); // d for decimal
fprintf(stderr, “%o\n”, ptr[8]); //o stands for octodecimal
fprintf(stderr, “%x\n”, ptr[8]); // x for hexadecimal
```

Output:
C
67
103
43

Set ptr[8] = 255; // ff ⇐== Super-important.

Another ff in the address. We’re figuring out the relationship between the input numbers and the crash site. This is what you can use to debug!

hexdump function: print out contents of memory

```
char ch = ‘A’;
hexdump(strderr, &ch, 1);
```

&: address of operator; it takes an object and turns it into a pointer (the C representation of an address) [reference]
*: address => object [dereference]

Output of hexdump:
address l hexidecimal l character correspondence

```
#include <stdlib.h> //need to include the library that has malloc
char ch_global = ‘B’;
const char ch_constant = ‘C’;  // const means this variable can never be changed
int main() {
        char ch = ‘A’;
        char* ch_heap = malloc(1);
        *ch_heap = ‘D’; //the lifetime of this object is independent of the running of the program.
```

The function that created it can return (i.e. end), and the dynamically-allocated object will continue to exist until it is freed.

```
        // C&P hexdump
fprintf(stderr, “%c\n”, ch);
```

```
hexdump(stderr, &ch, 1);  // stderr prints the output of the function to other
hexdump(stderr, &ch_global, 1);
hexdump(stderr, &ch_constant, 1);
hexdump(stderr, &ch_heap, 1);
}
```

There's only one version of the global variable for every program.

Heap variables are dynamically allocated: the lifetime of the variable that was allocated is independent of the function structure of the program.

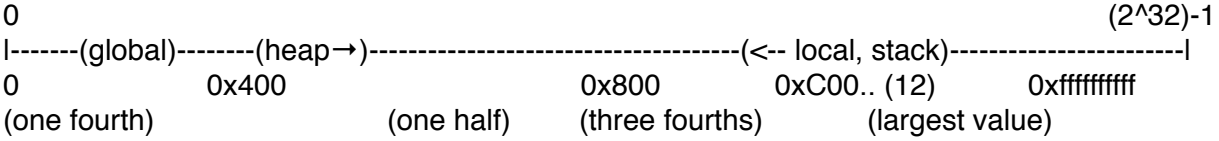
The object lasts between malloc and free (irregardless of returned functions).

(remember to include <stdlib.h>, which defines malloc)

Expectation: ABCD

B and C are right next to each other (addresses were 0804a040 and 08048b48, respectively); A looks "enormous" (address was bfc08143); D looks far away, but not that far away.

In address land: address run from 0 to ffffffff (2^32-1).



local variable near 0xC00.. (Do change: are in the stack)
global variables near 0x080… (Do not change)
heap variables near 0x090…. (Do change: are in the heap)

*note stack and heap addresses change because when they do change it is harder to hack into your computer

x86: name of the architecture of Intel chip
The first Intel chip that matters to us is the 8086, second 8186, …, Pentium Pro
x86 (because x is a variable)

Local variables are stored on the stack; they are stored roughly 3/4ths of memory and grown downwards (towards 0x00)

Global variables start close to 0.

Heap variables start close to 0 as well, and grow upwards (towards the stack)

What happens when they crash into each other? seg fault!

Object: local variable, global variable, something allocated dynamically

The compiler defines that every object is in separate memory (with a few exceptions), objects are stored in separate locations.

Program behaves as if we have an infinite amount of memory. But this is not actually true. The compiler is not allowed to let things overlap, so it will cause a segmentation fault.

We saw a stack overflow already when we had an infinite recursive stack.
The **abstract machine** says that memory will never overlap. However, in the **real machine** we see that heap values will fill up the heap and local values will fill up the stack.
(Cannot assign a value to a constant variable)

// trying really hard to set the constant
* (char*) &ch_constant = 'E'; // casting, telling the compiler to not report an error

What's in between the heap and stack? How can we find out?

hexdump(stderr, (&ch + &ch_heap), 1) // error

We can't add pointers in C, BUT we can subtract pointers and we can add numbers.

Still won't compile though because we can't subtract pointer types that aren't the same.

&ch_heap + (&ch - (char *) &ch_heap) / 2,
ch_heap is a variable on the stack
hexdump prints what's at the address passed in
hexdump(&ch_heap) // print the entire value of the pointer address of ch_heap (in reverse)

ch_heap + (&h - ch_heap) / 2 // core dumps

Bottom line: Nothing between the heap and the stack! (except maybe dragons)

Summary:

We began with a metaphor of a postman in a post office.

However, if the post office touches many of the boxes he dies!

Why?

Because, there are other people in the world who are spambots; lots of CS folks trying to stop them.

Best way to stop them: As soon as a program is going wrong - stop them! (why seg fault happens and why novice C programmers hate it)…

Good property for improved protection of your programs

Problem Set: You will create a better memory allocator that helps detect when you access invalid programs.

let's gooooo

Love,

Evan, Stephen, Ruth