C, Pointers, gdb

6.S081 Lecture 2

Fall 2020

My First Memory Bug

```
one = 1

two = one

two += 1

print(one)

print(two)

-----------------

1

2
```

```
abc = [‘a’, ‘b’, ‘c’]

abcdef = abc

abcdef += [‘d’, ‘e’, ‘f’]

print(abc)

print(abcdef)

-----------------

[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]

[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
```

Memory in C

Static Memory

- Global variables, accessible throughout the whole program
- Defined with *static* keyword, as well as variables defined in global scope.

Stack Memory

- Local variables within functions. **Destroyed** after function exits.

Heap Memory

- You control creation and destruction of these variables: *malloc(), free()*
- Can lead to memory leaks, use-after-free issues.

Pointers in C

A pointer is a 64-bit integer whose value is an *address* in memory.

Every variable has an address, so every variable’s pointer can be accessed, including a pointer to a pointer. And a pointer to a pointer to a pointer. And so on.

A pointers can handle arithmetic with the operators ++, --, +, -.

Pointer Syntax

int x = 5;

int *x_addr = &x; (same as int* x_addr = &x;) -> ex: 0x7ffd2766a948

***x_addr = 6;** -> you can use the * operator to access the underlying value.

int x_value = *x_addr; *dereferencing -> this gives 6*

int arr1[10]; -> Arrays are secretly pointers! More on that later.

int *arr2[20]; -> Array of pointers, making arr2 a pointer to a pointer.

void *myPtr;

Try these out! Make a new *user/* program like in Util.

Back to Memory

```
char* makeABC() {
  char y[3] = {'a', 'b', 'c'};
  return y;
}
```

What’s wrong with this?

Pointer Arithmetic, yay!

Suppose we have some **char *c** with value 0x100002.

c++; -> 0x100003

c += 4; -> 0x100007

Makes sense!

Pointer Arithmetic, sigh.

Suppose we have some **int *i** with value 0x100002.

i++; -> 0x100006

i += 4; -> 0x100016

Pointers add and subtract in increments of the base data’s length (in bytes).

Arrays in C

C arrays are contiguous blocks of memory holding a particular data type. The variable is the pointer to the beginning of the array.

char myString[40]; -> type of myString is char*

char* myArrayOfStrings[20]; -> type of myArrayOfStrings is char**

int counting[5] = {1, 2, 3, 4, 5}; -> type of counting is int*.

Arrays in C

The bracket operator (i.e accessing arr[1]) is just syntactic sugar for pointer arithmetic.

If we have **int arr[4] = {5, 6, 7, 8};** these are equivalent:

arr[2] = 50;

***(arr + 2) = 50;** -> Remember pointer arithmetic!

2[arr] = 50; -> Addition is commutative!

Arrays in C, Downsides

We are allowed to access or modify illegal memory by accessing an array out of bounds. C provides no checking whatsoever.

The behavior can be unexpected.

Use your size variables whenever possible!

Bitwise Operators in C

Everything is ultimately bits, C lets us manipulate those bits.

The following numbers are all binary:

& (and): **10001 & 10000** -> 10000

| (or): **10001 | 10000** -> 10001

^ (xor): **10001 10000** -> 00001

~ (complement): **~10000** -> 01111

Bitwise Operators in C

<< (left shift):

1 << 4 -> 10000 (binary) -> 16 (decimal)

>> (right shfit):

10101 >> 3 -> 10 (binary)

Bitwise Operators in xv6

We can combine these operators to make flag setting easy:

Define bit offsets **flag0 = 0, flag1 =1, flag2 = 2.**

To set flag **flag0** and **flag2:**

flags = (1 << flag0) | (1 << flag2) -> 101

To check if a flag is set in a **flags** integer:

if(flags & flag1) -> 101 & 010 == 0 (false!)

Casting in C

To cast in C: (newType)variable

void* to char*: **(char*)myVoidPtr**

uint64 from expression: **(uint64)(2 + 3), (uint64)myVoidPtr**

Casting in xv6

See kalloc.c and vm.c for some good examples.

```
extern char end[]; // first address after kernel.

void kfree(void *pa) {

  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)

    panic("kfree");

...
```

#include in C

.h files contain declarations (specs)

.c files contain definitions (implementations)

Basically never #include a .c file!

Include Guards help deal with nested/duplicate #includes (not used that much in xv6)

Use the **extern** keyword! Extends function’s visibility to all files in the program.