Section Notes 1 (Worksheet)

Setup

Resources

printf, C variable types/declarations, etc: http://cs61.seas.harvard.edu/wiki/2016/Resources

Infrastructure Issues?

http://cs61.seas.harvard.edu/wiki/2016/Infrastructure

Git

Make sure you can:

1. Pull the handout code from GitHub
2. Make a commit (let's edit the README!)
3. Push to your repo

For section, clone the cs61-sections repo so you can play with the code locally:

```
git clone git@github.com:cs61/cs61-sections.git
```

See here for more information: http://cs61.seas.harvard.edu/wiki/2016/Git

---

Assignment 1: Debugging Malloc

Motivation

Let's write some evil programs! List some possible memory bugs:

Memory bugs:

a.
b.
c.
d.
e.
f.

The Consequence of Memory Bugs: Undefined Behavior

What's wrong with a memory bug? Well, most memory bugs are completely evil, because they break the rules for correct C programs in a terribly consequential way. A memory error invokes what's called *undefined behavior*. Whenever a C program will definitely invoke undefined behavior, the program has *no meaning* any more—it's not even really a C program!—and it's allowed to do anything: crash, erase your hard drive, destroy your computer hardware, send the

contents of your files to Martian invaders, open up a backdoor for hackers, or even make demons fly out of your nose. Undefined behavior is a consequence of how close C allows programmers to get to computer hardware. Higher-level programming languages have no similar concept; if a program is erroneous, it crashes in a *defined* way. Modern C programmers must be aware of undefined behavior and avoid it at all times. We'll hear more about this in lecture.

Some Small Examples

Go to cs61-sections/s01. For each of the membug*.c files, name the memory bugs present in each file. Before running, predict the effect of the memory bug -- will the program crash? Fail silently? Run "make", then "./membug1", "./membug2", etc. to observe the effects of each bug.

a. membug1.c:
b. membug2.c:
c. membug3.c:
d. membug4.c:
e. membug5.c:
f. membug6.c:
g. membug7.c:

A bigger example: a buggy heap.

A heap is a tree-based data structure that satisfies something called the *heap property*. This property states that all pairs of parent/child nodes are ordered in the same way. For example, in a max heap a parent node is always greater than all its children. Therefore, you can see that the maximum element in a max heap is the root node of the tree. Heaps have O(log(n)) insertion and deletion time for their elements.

The code heap.c will read in integers as command line arguments and build a valid max heap from these numbers. Run "make" and then "./heap 3 4 5". It prints out the resulting heap; the max element should be first, and then the following rows print lesser items. We get this:

```
$ ./heap 3 4 5
5
4   3
```

This looks good—the maximum item is on the first line, and the other items are there too. But the next line doesn't look good:

```
*** Error in `./heap': munmap_chunk(): invalid pointer: 0xbffbde0c ***
Aborted (core dumped)
```

Core dump!! Time to boot up GDB to figure out what happened.

Bug 1

- Open up GDB with the command “gdb heap”. You can also pass in the -tui flag (“gdb -tui heap”) in order to have a graphical interface to see the code alongside the debugger.
- Run the program by typing “run 3 4 5”. GDB conveniently stops where the Segmentation Fault occurred.
- We can print the contents of all the variables in scope and try to figure out what went wrong. Do you see what happened? What is the bug in the code that caused this to occur?

There are lots more bugs in this code - see how many more you can find on your own! We will pick up and continue with this example next time.

Writing our own debugger

Open m61.c in the pset1 directory. Discuss the functionality of m61_malloc, m61_free, m61_calloc, m61_realloc. Open m61.h in the pset1 directory. Discuss the interaction between m61_malloc and base_malloc function calls (ex: calling base_malloc without M61_DISABLE means base_malloc calls m61_malloc). What can we add to m61_* implementations to avoid and keep track of these memory bugs?

Pointer Review

Given the following definitions what does each function do, and what do they return?
(Assume the arguments are cast to the right types)

1	`int sum(int a, int b) {`
	`  return a + b;`
	`}`
		`sum(8, 6) = ?`

2	`char* sum(char* a, int b) {`
	`  return &a[b];`
	`}`
		`sum((char*) 8, 6) = ?`

3	`int* sum(int* a, int b) {`
	`  return &a[b];`
	`}`
		`sum((int*) 8, 6) = ?`

Do functions 2 and 3 invoke undefined behavior?
Why do functions 2 and 3 have a different return value even though they look very similar?

Memory Layout

Top: memory location – This is the address where the bytes reside.
Middle: data – the bytes being pointed to.
Bottom: logical array index.

0x006	0x007	0x008	0x009	0x00A	0x00B	0x00C	0x00D	0x00E	0x00F
0x8B	0x14	0x24	0x08	0x05	0x44	0x24	0x04	0xC3	0x00
-2	-1	0	1	2	3	4	5	6	7

1. 'a' from example #2 is a memory address, so it is represented in the top, red row. The value of 'a' from that example is 8, so 'a' is 0x008, the third memory location from the left. Explain using the rows above how we got from 'a' to the final value of 14.
2. What is the value of '*a'?
3. How about a[6]?
4. Can you think of another way to write the function in example #2 which returns a char*? Hint: how are arrays and ptrs related?

More weird code syntax: what is the difference between a[5] and 5[a]?

Although it looks a little funky, a[5] is equivalent to *(a+5) in C. Similarly, 5[a] is equivalent to *(5+a)! (That's not so in Java, Javascript, or even C++.)

Pointer Equivalence

You can think of comparisons in three ways: data/value, point to the same memory, are the same memory.

If we have:

```
int c, d;
int *e, *f;
c = 10;
d = 10;
```

clearly c == d because we are comparing values, specifically the bytes 0x0A with 0x0A.

If we then say

```
e = &c;
f = &d;
```

then *e == *f, since 10 == 10, but e != f. Why??

Pointer comparison boils down to comparing the underlying addresses. Since the pointers point to different objects—e points to c, and f points to d—and C guarantees that different objects have distinct, non-overlapping addresses (with the exception of unions), we can see that e != f.

Equivalence Exercises

```
int a = 5;
int b = 5;
int* x = &a; // x==0xf4dc
int* y = &b; // y==0xf4d8
int* z = x;
int** p = &z; // p==0xf4c4
int array[10] = {5};
int yarra[6] = {1, 2, 3, 4,
5, 6};
int* w = array + 4;
int* group[3];
group[0] = array;
group[1] = yarra;
group[2] = y;
```

```
1. (a == b) ? True : False;
2. (x == y) ? True : False;
3. (y == &a) ? True : False;
4. (*z == a) ? True : False;
5. (*group[0] == a) ? True : False;
6. ((*group)[4] == (*(group+1))[4]) ? True :
False;
7. ((*group)[0] == (*(group+1))[4]) ? True :
False;
8. (*w == 4) ? True : False;
9. (*w == 0) ? True : False;
10. (w == 0) ? True : False;
11. (((*(group+1))[3]) == 1) ? True : False;
12. ((**(group+2)) == 5) ? True : False;
```

Symbol	Type	Value
z		
(&z)-2		
yarra[3]		
&b		
group[2][0]		
array[6]		
*p		
**(x-3)		
**(&(group[1])-1)		
*w		
group+2		
*(group+3)		

Assignment 1 Hints

1. Use structs to avoid complicated pointer arithmetic.

Suppose ptr is a pointer to the start of where we are placing our metadata, and we want to place the following types in the metadata: int, int, char*. Let's look at this code:

```
// declare metadata values
int a = 5;
int b = 6;
char* c = // some pointer

void* ptr = // pointer to beginning of metadata;
*((int*) ptr) = a;
*(((int*) ptr) + 1) = b;
*(((char**) ptr) + 2) = c; // char * is 4 bytes on 32-bit machine
```

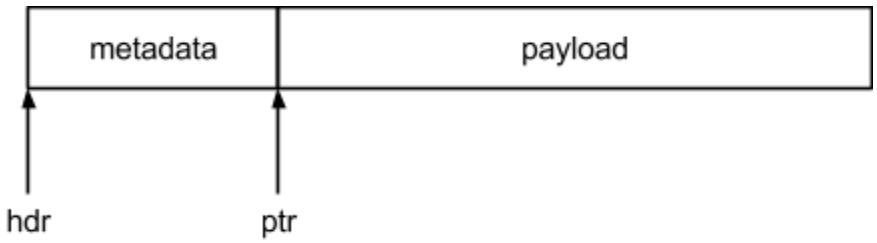
This is horrendous code. There is a much more elegant and robust solution using structs:

```
struct header {
  int a;
  int b;
  char* c;
};

void* ptr = // pointer to beginning of metadata;
struct header* hdr = (struct header*) ptr;
hdr->a = a;
hdr->b = b;
hdr->c = c;
```

2. Avoid pointer arithmetic on (void*) types!

Suppose ptr is a pointer to the beginning of the payload, and we want to move the pointer back to get to the beginning of the metadata:



```
void* ptr = // pointer to beginning of payload
struct header* hdr = (void*) (ptr - sizeof(struct header));
// avoid this!!! void* arithmetic is not well defined;
// this works on GCC and Clang in some modes but not in others
```

Instead:

```
void* ptr = // pointer to beginning of payload
struct header* hdr = ((struct header*) ptr) - 1; // much more robust
```

Fun Exercise: Identify the Undefined Behavior!

```
int lotsOfUndefinedBehavior(void) {
    int i = 0;
    int j;
    char* message = "Hello World";
    message[0] = 'J';
    printf("%s", message);

    i++;
    printf("New value is: %d", i);

    j++;
    printf("New value is: %d", j);

    char* p = (char*) malloc(sizeof(char)/sizeof(int));
    p[0] = 'a';
    printf("Our string is: %s", p);

    int* nats = (int*) malloc(sizeof(int)*42);
    nats[0] = 1;
    For (int k; k < 43; k++) {
        nats[k] = 2*nats[k-1];
    }
}
```