Lecture notes: January 30, 2017

Topics:

1. More on memory allocation
2. Memory leaks
3. Valgrind

Readings: Forouzan and Gilberg, 10.4

Allocating arrays of structs:

When we want to allocate an array of 15 integers, we might do something like this:

```
int * p = malloc (15 * sizeof(int));
```

What if we want to allocate an array of 15 point structures?

```
typedef struct {
  float x;
  float y;
} Point;
```

We can do the same thing!

```
Point * p_array = malloc(15 * sizeof(Point));
```

Now `p_array` points to an array of 15 `Point`s, and we can access, say, the 10th `Point` like so: `p_array[9]`

Memory leaks:

Remember that calling `malloc` grabs a block of memory in the heap, and `malloc` guarantees that this block of memory won't be used by anyone else:

```
int * p = malloc(10 * sizeof(int)); //grab 40 bytes of memory
```

To give a block of memory back to the heap, so that it can be reused by something else, we call `free`:

```
free(p); //release 40 bytes of memory back to the heap.
```

So what happens if you call `malloc` inside a function?

```
void foo(int N) {
  int * p = malloc(N * sizeof(int)); //allocate an array of N integers
  ... //code to do something with the array
  return;
```

```
}
```

The local variable `p` points to the block of memory we grab with `malloc`. But once we return from the function, the variable `p` goes away. We no longer have any way of *reaching* the block of memory we allocated: we have no way of getting to the address of that block. If we can't get the address of the block, we can't `free` it — we can never give the memory back to the heap. This is a *memory leak.*

If we keep calling `foo`, we'll keep allocating new blocks of memory that we lose access to. These blocks can never be given back to the heap and will just sit around taking up space. In fact, if we call `foo` too many times, we could run out of memory!

Memory leaks are bugs in your program, though they may not always cause your program to break. What you need to do is make sure that whenever you are done using a block of memory you have allocated, you call `free` on it:

```
void foo(int N) {
  int * p = malloc(N * sizeof(int)); //allocate an array of N integers
  ... //code to do something with the array
  free(p); //free the array to avoid a memory leak
  return;
}
```

But finding memory leaks can be tricky: you have to free memory late enough that you're sure you're done with it, but *early* enough that you still have access to it. The previous example of a memory leak was an example of waiting too long to free memory: by the time we returned from `foo`, we didn't have a way of getting to the block of memory that we wanted to free. Here's an example where we need to be careful not to free memory too early:

```
int * foo(int N) {
  int * p = malloc(N * sizeof(int)); //allocate an array of N integers
  ... //code to do something with the array
  free(p); //THIS IS TOO EARLY!
  return p;
}
```

Why is it a problem to free the array in this example? Because we've returned the pointer to the array. That means we can't be sure that whoever *called* `foo` isn't going to use the array for something else. If we free the array in `foo`, we are returning memory that the program might not be done with. That's bad!

Some times it can be even harder to tell that whether it's safe to free memory:

```
int * * bar(int N) {
  int * p = malloc(N * sizeof(int)); //allocate an array of N integers
  int * * q = malloc(sizeof(int *)); //allocate space for an int*
  * q = p; //the box q points to now holds the address of the array
  return q; //return the address of the box q points to.
}
```

It's not safe to free the array before returning from `bar`. Even though we're not returning a pointer to the array (like before, in `foo`), we're returning a pointer *to* a pointer to the array. Which means we can still *reach* the array, and freeing the array is unsafe. But it gets worse:

```
int * * i = bar(10); //i points to a box which points to the array
(* i)[5] = 12; //this sets the 6th element of the array.
(* i) = NULL; //now i points to a box which contains NULL
```

Once we execute the third line of code, we cannot reach the array any more! The array has leaked!

To fix this, we need to make sure to free the array before we make it unreachable:

```
int * * i = bar(10); //i points to a box which points to the array
(* i)[5] = 12; //this sets the 6th element of the array.
free(* i); //free the array
(* i) = NULL; //now i points to a box which contains NULL, but it's OK
```

Valgrind

How can we find memory leaks? They're not really like other bugs. Some bugs show up at compile time because the compiler complains. Other bugs cause your program to crash. But memory leaks some times don't seem to affect your program at all! They can be hard to find. We can use a tool called *valgrind* to help find bugs. Consider the following program:

```
void main () {

  int * p = malloc(sizeof(int));

  int * q = malloc(4 * sizeof(int));

  int * r = malloc(16 * sizeof(int));

  free(p);

  free(r);

}
```

This program has a memory leak (we don't free q before the program exits). If we compile the program and run valgrind:

```
> gcc alloctest.c -g
> valgrind --tool=memcheck --leak-check=full a.out
```

We get the following output, telling us that we leaked 16 bytes. Valgrind also tells us which call to malloc caused the leak (line 5, which is the allocation for q).

```
==14189== Memcheck, a memory error detector
==14189== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==14189== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==14189== Command: a.out
==14189==
```

```
==14189==
==14189== HEAP SUMMARY:
==14189==     in use at exit: 16 bytes in 1 blocks
==14189==   total heap usage: 3 allocs, 2 frees, 84 bytes allocated
==14189==
==14189== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==14189==    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==14189==    by 0x400523: main (alloctest.c:5)
==14189==
==14189== LEAK SUMMARY:
==14189==    definitely lost: 16 bytes in 1 blocks
==14189==    indirectly lost: 0 bytes in 0 blocks
==14189==      possibly lost: 0 bytes in 0 blocks
==14189==    still reachable: 0 bytes in 0 blocks
==14189==         suppressed: 0 bytes in 0 blocks
==14189==
==14189== For counts of detected and suppressed errors, rerun with: -v
==14189== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```