CSCI 1300

Intro to Computing

Gabe Johnson

Lecture 35 Apr 12, 2013

Linked Lists

Lecture Goals

1. Pointers/Structs/Arrays HW
2. Linked Lists

Upcoming Homework Assignment

HW #9 **Due: Friday, Apr 12**

Pointers, Arrays, Structs

The HW on pointers is due tonight. You can get 5 points extra credit on this one. Will take questions about how to do things, and why some things screw up.

HW Status

```
0: 21    | **********************
1: 0     |
2: 1     | *
3: 0     |
4: 8     | ********
5: 3     | ***
6: 1     | *
7: 0     |
8: 0     |
9: 19    | ********************
10: 0    |
11: 0    |
12: 38   | ****************************************
13: 0    |
14: 0    |
15: 65   | *****************************************************************
Total students participating: 156
5 students doing project
```

Returning ptr to stack mem

```
int* get_array(int n) {
    int ret[n]; // allocate stack memory
    return ret; // return pointer to it
}
```

When the above function returns, the memory that stores the ‘ret’ array is marked as fair game for re-use. So even though we have a pointer to that memory, the contents of that memory might change without notice.

Returning ptr to heap mem

```
int* get_array(int n) {
    int* ret = new int[n]; // heap memory
    return ret; // return pointer to it
}
```

The solution is to allocate dynamic memory on the heap. The computer will not recycle this until it is explicitly deleted (using the ‘delete’ keyword.)

std::bad_allocator

One common problem:

```
int* ret = new int[n]; // make dynamic int array
```

... but what if **n** is negative? “Yo computer, give me negative 200 bytes!” ... “aaaeeeeiiiiiiiii!”

When computers freak out they do it *methodically* (usually). In this case it *throws an exception* called *bad_allocator*.

Exceptions

You don’t really need to dive deeply into what exceptions are. Suffice it to say they are a form of runtime problem that the computer recognizes and gives you a chance to recover from.

If you do a software engineering course, (or if you want to learn how to engineer), you’ll need to study up on these things.

Bad Burrito

```
burrito* b = new burrito; // heap mem for burrito
b->beans = 4.0;
b->rice = 5.0;
b->salsa = 3.0;
b->steak = 4.0;
return b;
```

If this is what your burrito function looks like, it might actually pass the tests on either your machine or on the RG server. It might even pass both. But it still has problems. Welcome to C and C++'s daily dose of *truth*.

Bad Burrito

```
burrito* b = new burrito; // heap mem for burrito
b->beans = 4.0;
b->rice = 5.0;
b->salsa = 3.0;
b->steak = 4.0;
return b;
```

Problem is: burritos have nine floating point variables inside them, five of which we did not initialize. So the memory for ‘chicken’ and ‘veggies’ and the like has *whatever garbage was left there*. Maybe 300. Maybe -2384234423. Maybe you get lucky and it is already 0.

Bad Burrito

```
burrito* b = new burrito; // heap mem for burrito
b->beans = 4.0;
b->rice = 5.0;
b->salsa = 3.0;
b->steak = 4.0;
b->peppers = 0;
b->cheese = 734;
b->pork = -8974123;
b->chicken = 29834;
b->veggies = 19823428394;
return b;
```

If you don't initialize your variables to something, it is sort of like letting the computer choose for you at random.

Bad Burrito

```
burrito* b = new burrito;
b->beans = 4.0;
b->rice = 5.0;
b->salsa = 3.0;
b->steak = 4.0;
return b;
```

But why does it work on one computer and not another?

This has to do with how two different computers handle memory recycling. If my computer allocates memory that was recycled a long time ago, I will get consistent (possibly consistently *wrong*) results. If the RG computer allocates memory that was just recycled, it will be much less consistent.