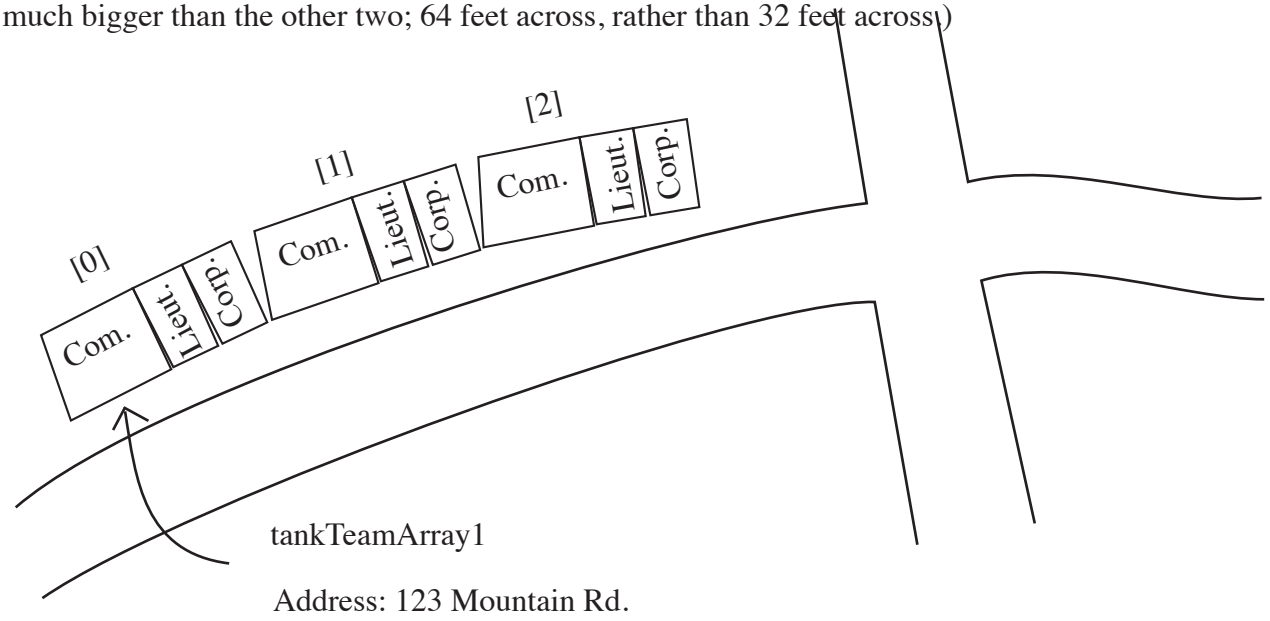
Memory and Arrays

Analogy: Teams of tank soldiers living in barracks on an army base.

Each unit is made up of a commander, a lieutenant, and a corprel.

At the base, each tank team of three lives in one housing unit, and all the housing units for the tank teams are next to each other . (The commander's apartment is much bigger than the other two; 64 feet across, rather than 32 feet across)



Because all the tank personelle live together, there is no need of addresses for each one. They are all able to be located relative to each other. The address of "tankTeamArray1" is 123 Mountain Rd.

Commander Chip Doehring lives in the third apartment. So to access him, we would go to the address 123 Mountain Rd., and then march 64+32+32 feet x 2 to get to his apartment.

If this were an array called tankUnit, here's what the code would roughly look like:

```
public class TankTeam(){
        private double commander;
        private int lieutenant;
        private int colonel;
...
}

TankTeam[] t1 = new TankTeam[3];
...
TankTeam[2] =
System.out.println(t1[2].getCommander);
...
```

So a key point with this arrangement is that we can locate any of the soldiers simply by knowing the address of the entire tankTeamArray1. But there are issues with this arrangement.

Limits to the Array arrangement:

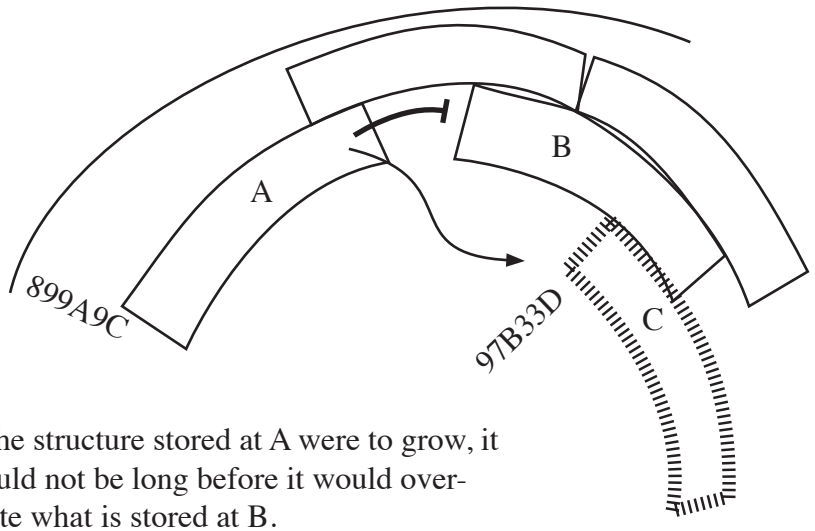
In terms of actual arrays in memory, if the elements were not next to each other, then they could not be found. The is because each array is only one address (at the beginning of the array). And this is so because of the prime advantage you noted the other day, that when passing an array as a parameter, we just copy a 32 bit addresss, not the entire array.

So the problem with arrays is we have to reserve the amount of memory for the absolute maximum number of elements we believe the array will ever take. And until it is full, it thus wastes memory.

If the army base analogy worked the same way, there would be lots of apartments made that would stand empty, waiting for them to possibly be filled.

Linked List Concept to the Rescue

So if we are to have a data arrangement where we can add onto a structure, keep in mind how memory is allocated; it is random, it is blind. When something is saved to a hard drive, or loaded into RAM, the algorithm is simply to find a free space with enough memory to accomodate what we are looking to save at present, without concern for the possibility that that structure might grow. So if it were to grow, contiguously, it could very well run into, and overright other data in memory. As thus:



If the structure stored at A were to grow, it would not be long before it would overwrite what is stored at B.

So if A is to be able to grow dynamically, it needs to pick another place that is free with enough space which is required. So A needs an address of where that free space begins..., perhaps at the area of memory labeled C.

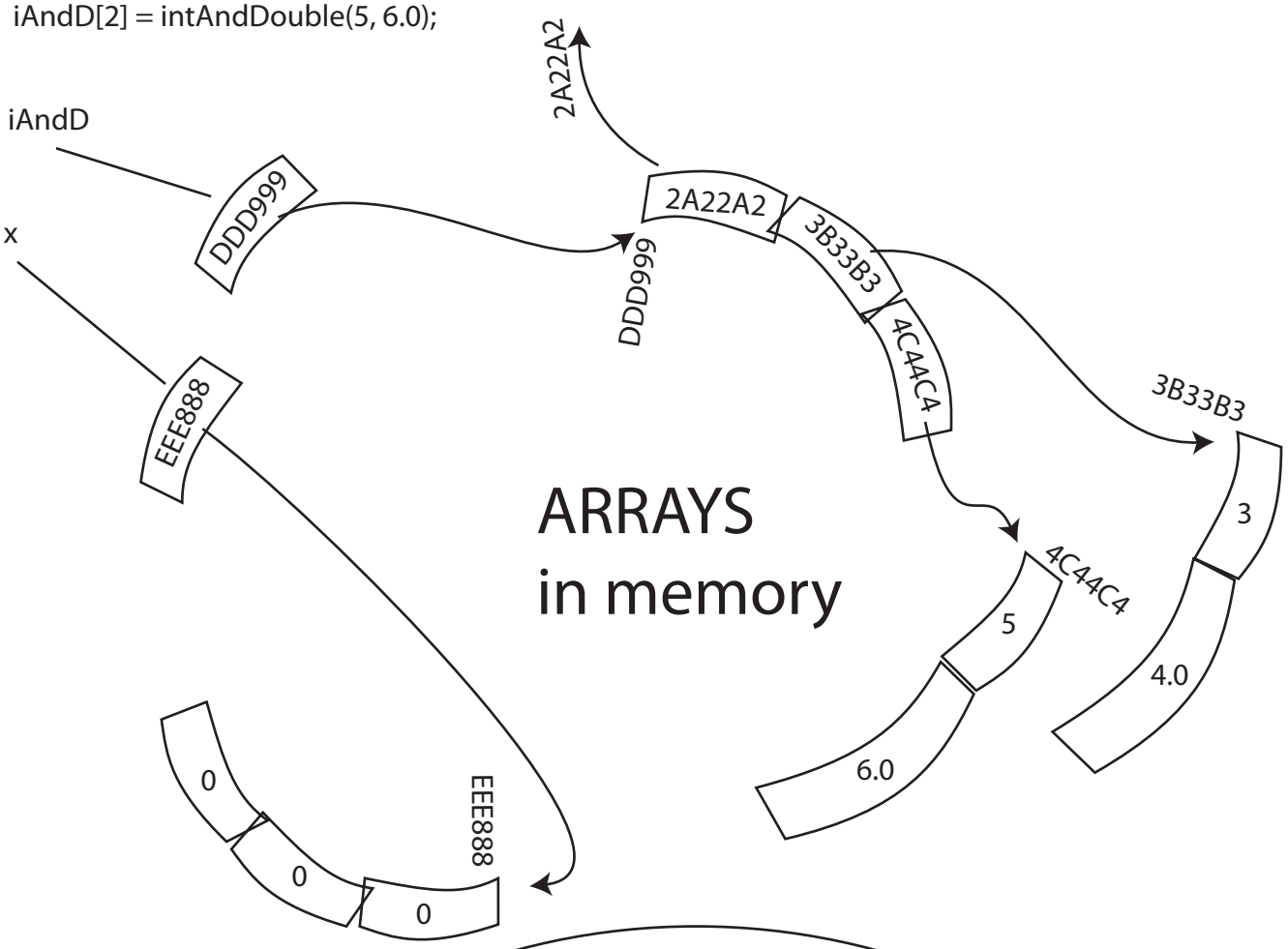
So, the last part of A, in order to grow, would have to be able to "point" to the memory address, in this case, 97B33D.

So that would make a structure whose name is a shortcut for memory address 899A9C, and which links to 97B33D, and so on. We'll end up with a list of things which is linked together to make the entire structure; a "linked list" in fact.

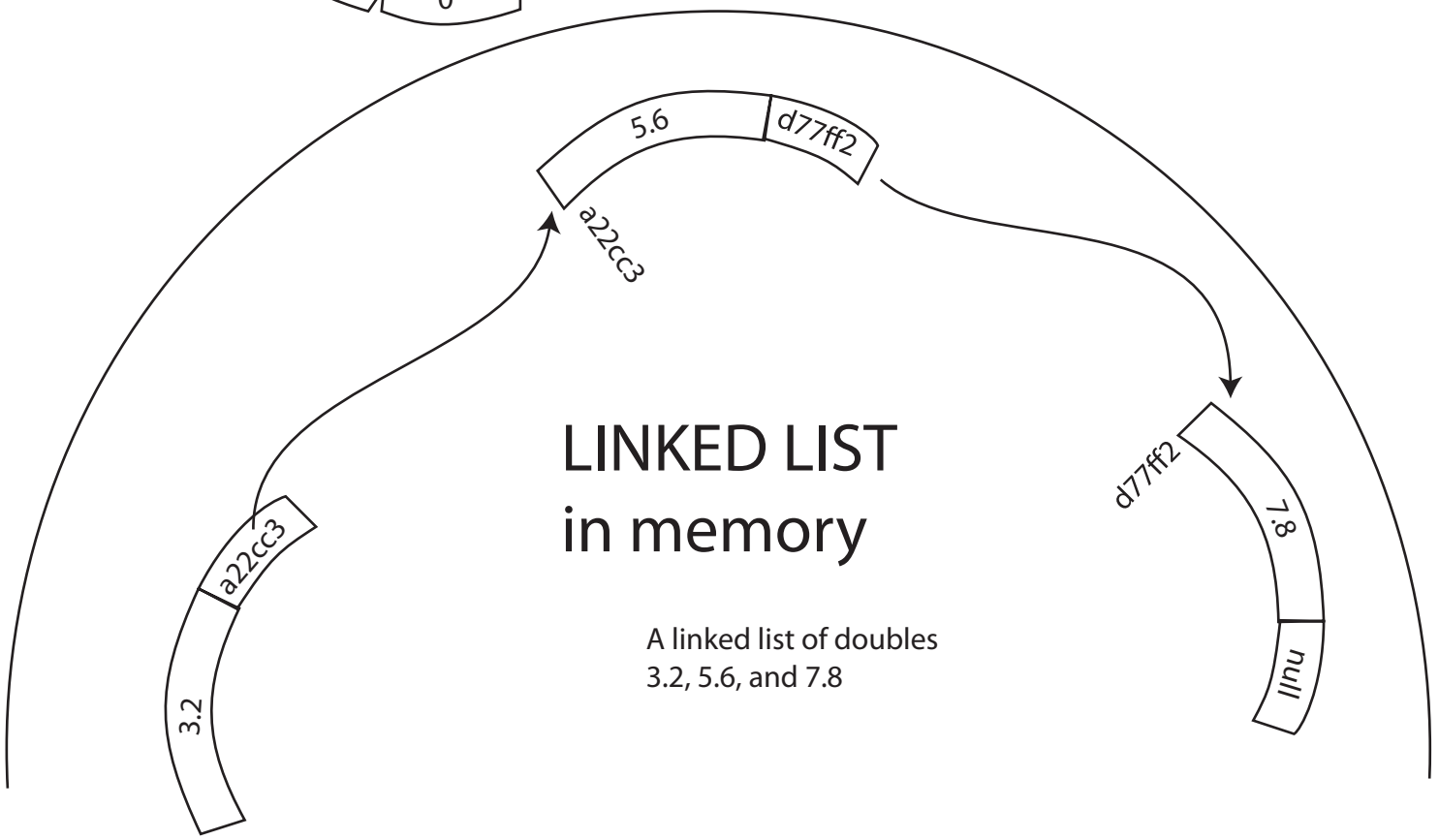
```
public class intAndDouble{
        private int i;
        private double d;
...}

intAndDouble[] iAndD = new intAndDouble[3];
iAndD[0] = intAndDouble(1, 2.0);
iAndD[1] = intAndDouble(3, 4.0);
iAndD[2] = intAndDouble(5, 6.0);
```

int[] x = new int[3]

1 2.0



ARRAYS
in memory



LINKED LIST
in memory

A linked list of doubles
3.2, 5.6, and 7.8