CS 2110 Lab 22
Shallow vs Deep Copy

The TAs :)

November 14, 2018

1 Recitation Outline

1. Shallow Copy
2. Deep Copy

2 Shallow Copy

Let's say we have a person struct defined in the following way:

```
typedef struct {
    int age;
    char *name;
} person;
```

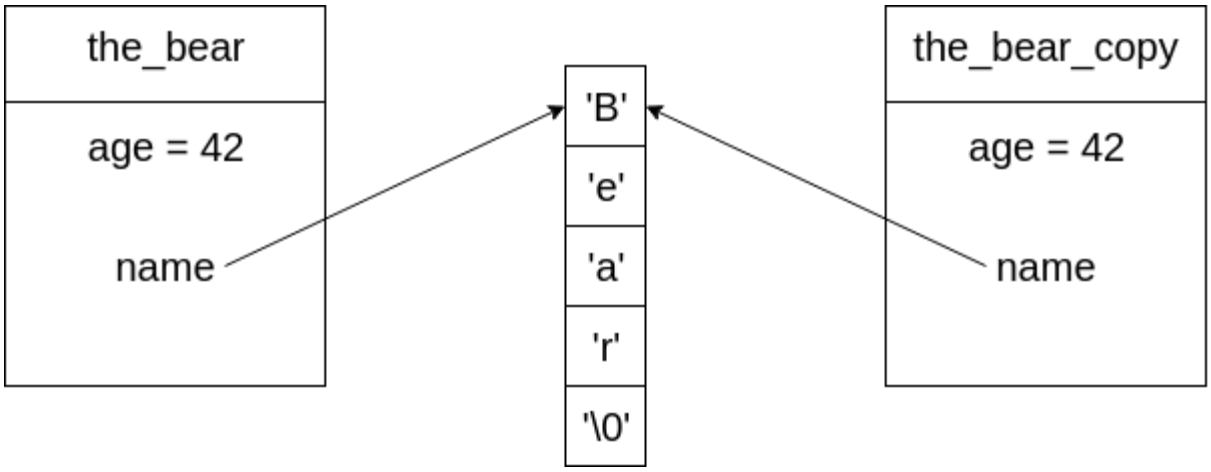
And we have a the following variable for a person:

```
person the_bear;
the_bear.age = 42;
char bear_name[5];
bear_name[0] = 'B';
bear_name[1] = 'e';
bear_name[2] = 'a';
bear_name[3] = 'r';
bear_name[4] = '\0';
the_bear.name = bear_name;
```

What we'll call a shallow copy of `the_bear` could be made like this:

```
person the_bear_copy = the_bear;
```

This would create a new struct with the same age and name pointer as the original `the_bear` variable. What this means though, is the name pointer for `the_bear_copy` is pointing to the **same** address as the name pointer for `the_bear`, so if we alter the name of one, both will see the change. Visually, you can think about it like this:



Why does this happen? Well when C is looking at the variable `name` it only sees a pointer, it has no idea if `name` is pointing to the start of an array or just to a single character. The only thing C does know for certain is that the variable `name` holds an address, which is where it's pointing to. So to be safe not to make an invalid assumptions, that's all C copies. This means the copied `name` is pointing to the same character array in memory as the original.

3 Deep Copy

What if we want to create a brand new character array for our new person that can be changed independent of the original? In this case we'll do what we'll call a deep copy.

First, we're going to create a shallow copy of our person the same way we did before:

```
person the_bear;
the_bear.age = 42;
char bear_name[5];
bear_name[0] = 'B';
bear_name[1] = 'e';
bear_name[2] = 'a';
bear_name[3] = 'r';
bear_name[4] = '\0';
the_bear.name = bear_name;

person the_bear_copy = the_bear;
```

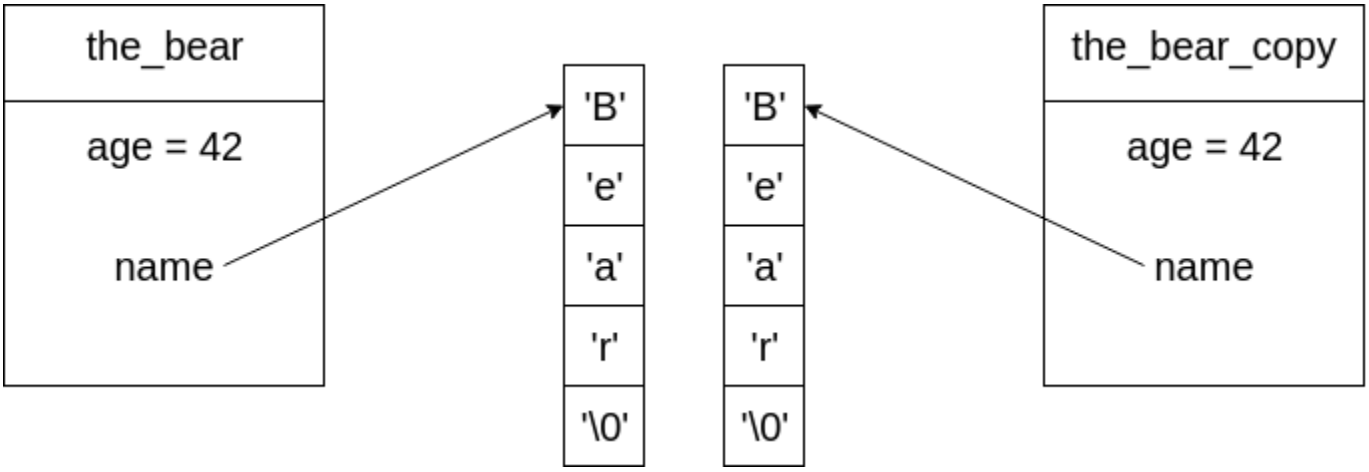
But now we want to allocate a new array for our name, and copy over each character from the old name one by one:

```
char bear_copy_name[5];
for (int i = 0; i < 5; i++) {
    bear_copy_name[i] = the_bear.name[i];
}
```

Now that we have a newly copied name array, we can assign it to the name variable of our copied person:

```
the_bear_copy.name = bear_copy_name;
```

Now we can safely modify the original or the copied person's names, without it affecting the other name! Visually, you can think about it like this:



The biggest takeaway from shallow vs deep copy is that a shallow copy copies pointers, while deep copy copies the actual elements in memory.

3.1 Using a fixed size array instead of a pointer

As an aside, we can get the same result of the deep copy, with the same logic as the shallow copy, but at a pretty big cost. How? Well what if we could give C more information about what `name` really is? Instead of declaring `name` as a char pointer, let's declare it as a char array of a fixed size!

```
typedef struct {
    int age;
    char name[5];
} person;
```

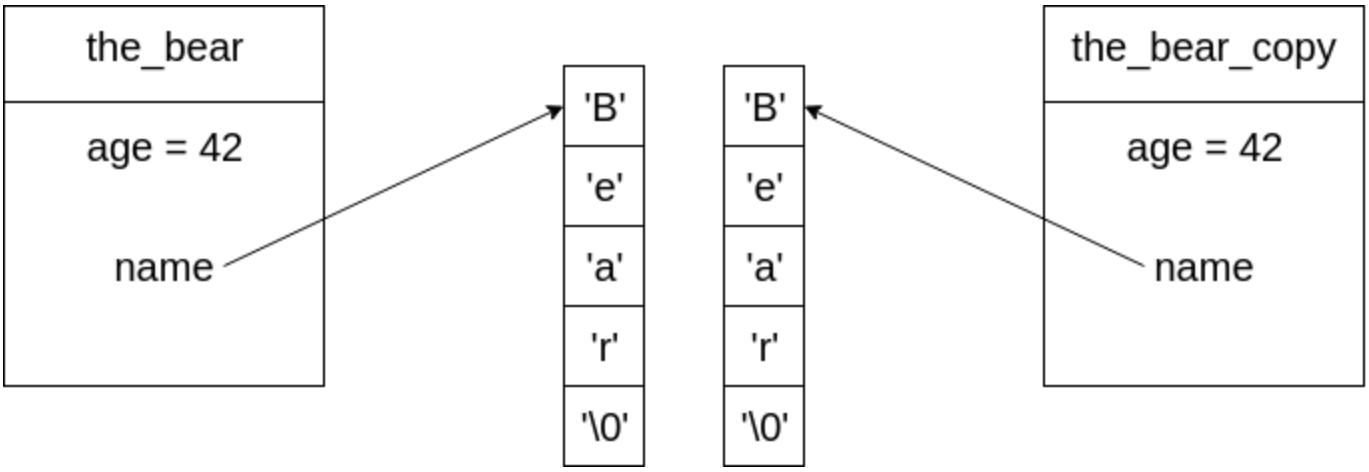
So we start out by declaring our first person (Note how we don't declare a new array after declaring the struct, this is because our struct creates an array of 5 elements for us!):

```
person the_bear;
the_bear.age = 42;
the_bear.name[0] = 'B';
the_bear.name[1] = 'e';
the_bear.name[2] = 'a';
the_bear.name[3] = 'r';
the_bear.name[4] = '\0';
```

Now if we do a shallow copy:

```
person the_bear_copy = the_bear;
```

When C looks at the name variable it knows it's pointing to the start of an array of 5 characters, so it can confidently copy all 5 of those characters as well. So `the_bear_copy` has a brand new name that can be modified independently of the original! This creates the same result as our deep copy approach:



This is great and all, but now we've had to restrict our name to only be 4 characters long! Since if we don't give C a fixed size, it doesn't know how many characters `name` is pointing to and so with our shallow copy code, it can't copy it over like we wanted.