

## Variables as Pointers

We'll take a deeper look at variables in this section, and specifically how they act as pointers to a place (or address space) in memory. This is a concept that confuses a lot of new programmers, and it's also one of the most important concepts to understand. The easiest way to understand variables and pointers is to look at some examples.

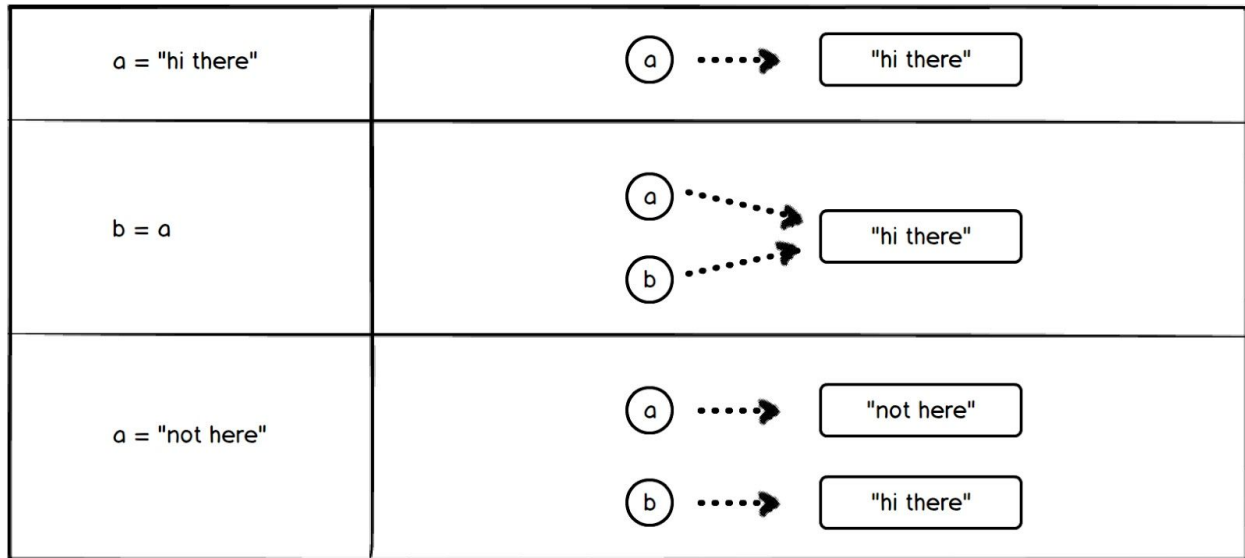
```
a = "hi there"  
b = a  
a = "not here"
```

What is `b` in the above code? Think about it for a second. What about the code below?

```
a = "hi there"  
b = a  
a << ", Bob"
```


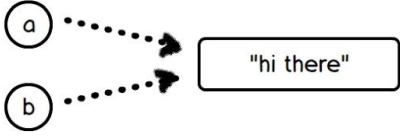

What is `b` in this example? If you tried both code snippets in `irb` (if you didn't, you should before continuing), you'll see that `b` doesn't reflect the value of `a` in the former example, but does so in the second example. Why is this?

To understand the difference in the above two code snippets, you need to understand that **variables are pointers to physical space in memory**. In other words, variables are essentially labels we create to refer to some physical memory address in your computer. In the first example, this is what happened:



From the above diagram, we can see that the code `a = "not here"` reassigned the variable `a` to a completely different address in memory; it's now pointing to an entirely new string. This is what the `=` operator does. It's important to understand that different memory space can in fact hold the same value, but they are still different places in memory. For example, if our last line of code was `a = "hi there"`, the result would still be the same: `a` and `b` in that case would still point to *different* addresses in memory; they would just happen to have the same value.

Now let's see what the second snippet of code did.

<code>a = "hi there"</code>	 <pre> graph LR     a((a)) -.-&gt; str1["hi there"] </pre>
<code>b = a</code>	 <pre> graph LR     a((a)) -.-&gt; str1["hi there"]     b((b)) -.-&gt; str1 </pre>
<code>a &lt;&lt; ", Bob"</code>	 <pre> graph LR     a((a)) -.-&gt; str2["hi there, Bob"]     b((b)) -.-&gt; str2 </pre>

Interesting! The line of code `a << ", Bob"` did **not** result in reassigning `a` to a new string. Rather, it *mutated the caller* and modified the existing string, which is also pointed to by the variable `b`. This explains why in this code, `b` reflects the changes to `a` - they're both pointing to the same thing.

This is the major point of this section: some operations will mutate the actual address space in memory, thereby affecting all variables that point to that address space. Some operations will not mutate the address space in memory, and instead will re-point the variable to a new address space in memory.

This also applies to variables that point to arrays, hashes, or any data structure that has methods that mutate the caller. If you call a method that *mutates the caller*, it will change the value in that address space, and any variables also pointing there will be affected. Therefore, always pay attention whether your variables are pointing to the same object (space in memory) or if they are dealing with copies that occupy different address space in memory.

Play around with some examples in `irb`. Some examples you can try are:

```
a = [1, 2, 3, 3]
b = a
c = a.uniq
```

What are `a`, `b` and `c`? What if the last line was `c = a.uniq!`?

```
def test(b)
  b.map {|letter| "I like the letter: #{letter}"}
end

a = ['a', 'b', 'c']
test(a)
```

What is `a`? What if we called `map!` instead of `map` from within the `test` method, how would that affect the value of `a`? Remember when we're passing in arguments to a method, we're essentially assigning a variable to another variable, like we did with `b = a`. Working with `b` in the `test` method may or may not modify the `a` in the outer scope, depending on whether we modify the address space in memory that `a` is pointing to.

This is all you need to know for now. It's almost guaranteed that you'll run into bugs in the future related to this topic, so the important thing here isn't to memorize the rules, but to understand the concept and be able to jump into `irb` to refresh your memory.

## Blocks and Procs