Pass by Value/Reference

To describe this, I'm going to use examples from C where this is a lot more explicit as well as it being a language you've seen before.

The call stack (or stack for short) is used to keep track of the execution state of a program. It is used for storing things with a known fixed-size *at compile-time*. Some examples of these sorts of types from languages you probably have seen:

C
int, double, char, float, struct
Java
int, double, char, float, short, long
Swift
structs, enums

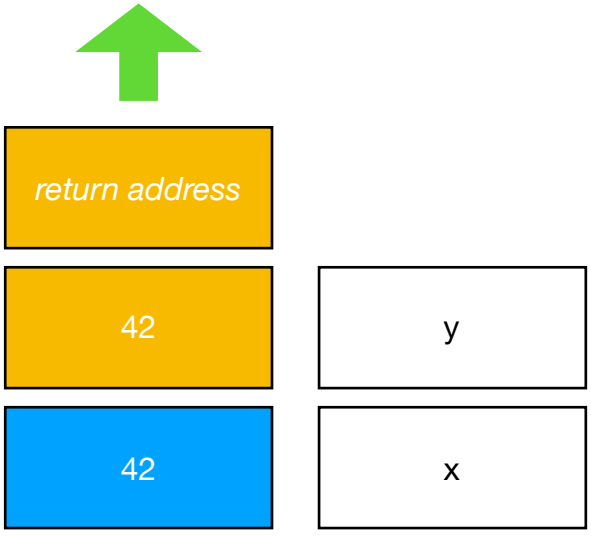
Note that the exact size of these things is dependant on the compiler of machine you're working on. I'll work on an example in C to illustrate what's going on.

Suppose we have a program that looks like the code to the right:

```c
#include <stdio.h>

void change(int y){
    int val = 3;
    y += val;
}

int main() {
    int x = 42;
    change(x);
    printf("%d\n", x);
}
```

1. Pass by Value

1.1 Before you go further, what does the program above output?



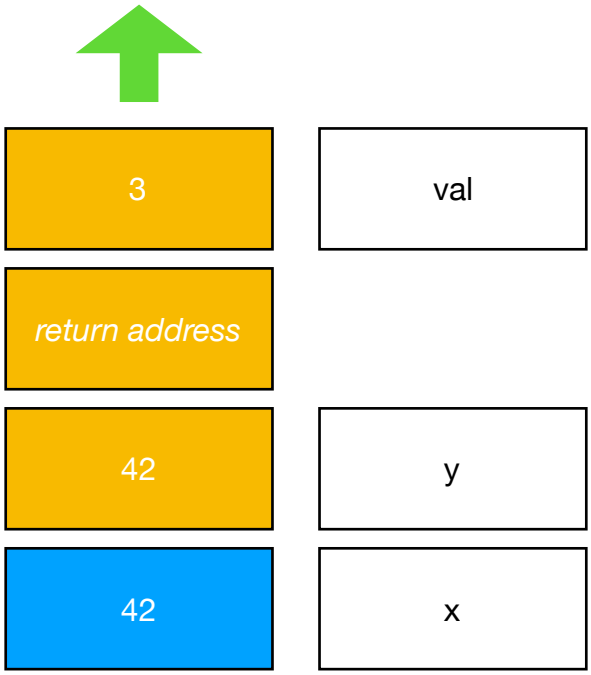
If you said '42', then you're correct!

Why does it do that? Well it's because the argument (42) is being ***copied*** into the call to change. When you call a function, the compiler needs to gather all the information together to pass to the called function. If we could examine the call stack just before actually calling change, we'd end up with something that looks like the diagram to the left.

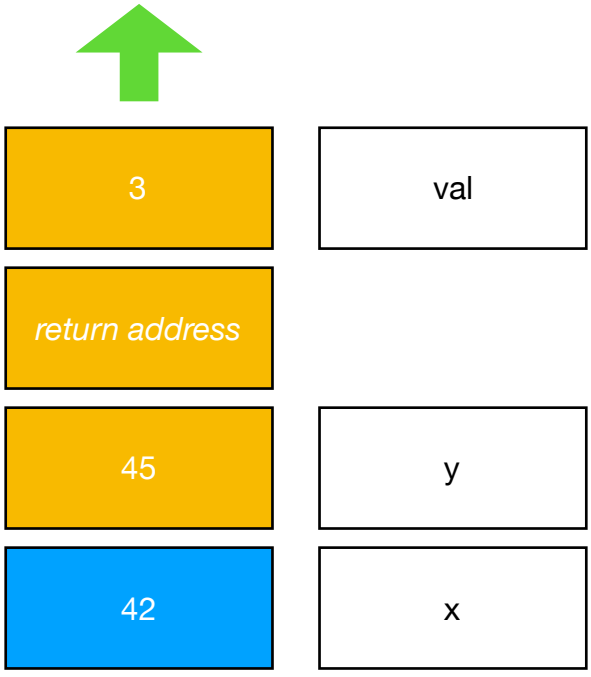
There are a couple of things to note about the diagram. I've drawn the stack growing upwards (indicated by the green arrow). In the white boxes to the right I've indicated the labels of the variable names. The *return address* is determined by the compiler and allows the change function to resume execution where it left off.

The yellow/orange boxes belong to the call to change, while the blue box is the local variables for main. The actual way this is represented in the computer is architecture and compiler specific.

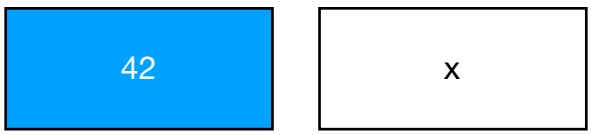
So, what happens when we actually execute the change function? Well we add the local variables to the top of the stack …



… and then execute the rest of the function by performing the addition …



… before we finally return.



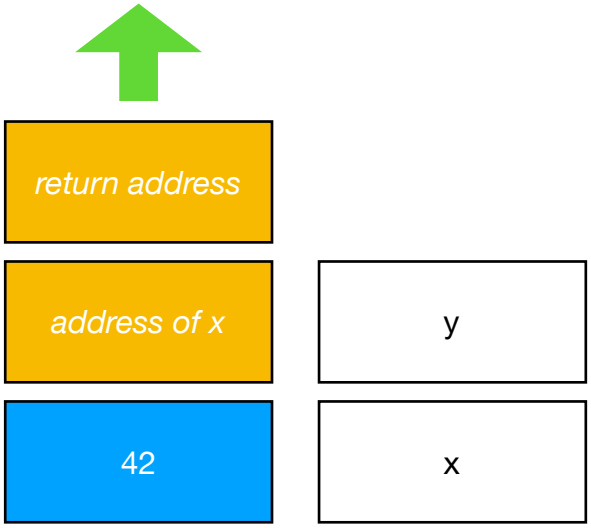
2. Pass by Reference

Suppose we make a small change to the original program so that instead of accepting a raw integer, we instead have a pointer to an integer, so that our program looks like the one on the right. What effect does that have?

2.1 What do you expect to be printed with this program?

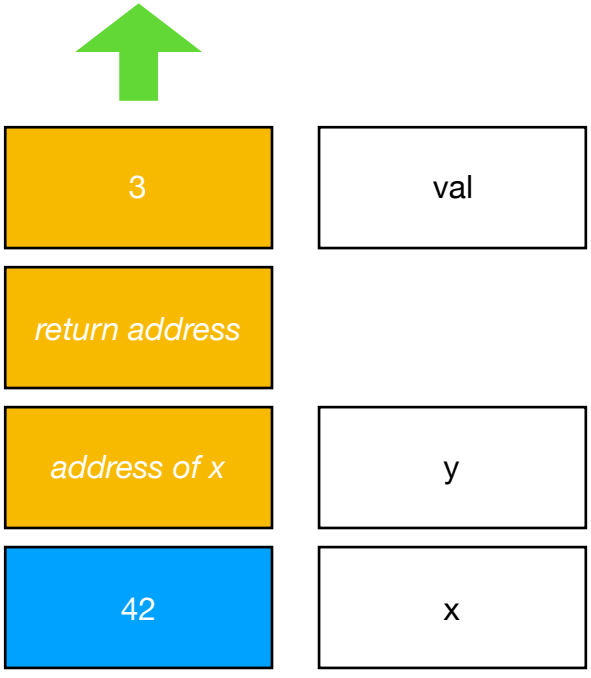
2.2 What do you think is going on with this program's call stack?
Draw the stages of it *before* you read on.

```
#include <stdio.h>

void change(int* y){
    int val = 3;
    *y += val;
}

int main() {
    int x = 42;
    change(&x);
    printf("%d\n", x);
}
```



I f
you said '45', then you're correct! What's going on here? Well it's almost the same thing as before, but we're now using a *pointer*.

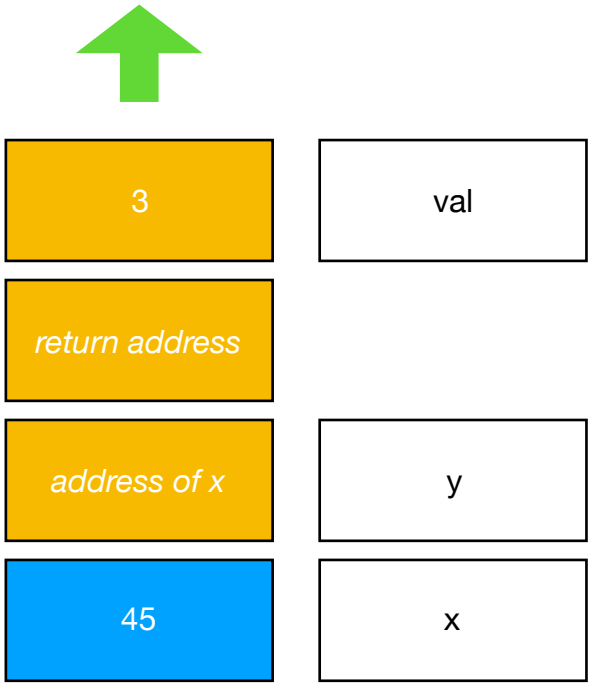
Before we execute the change function, we have a stack that looks like the one below on the left. Do you notice that y is now the address of x?



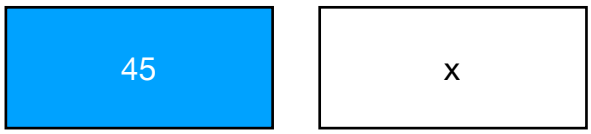
How does the rest of the execution play out? Let's have a look. We still setup out local variable as before. When we come to do the addition, we're saying to treat y as an address and to lookup the value stored there. This is what the *y is doing at the end of the change function.

Because we're treating y as an address, we're able to go to the original value (in the main function) and change it's value.

We then execute the rest of the function by performing the addition …



… before we finally return.



I want to make it clear here, that the address is still being copied but the way it's interpreted is different. It's not making a copy of the thing at that location.

2.3 When might this be a good thing to do?
2.4 What do the following snippets of code print?
2.5 Sketch the call stacks for these programs two.

```
#include <stdio.h>

void swap(int* y, int* z){
     int temp = *y;
     *y = *z;
     *z = temp;
}

int main() {
    int p = 42;
    int q = 24;
    swap(&p, &q);
    printf("%d -> %d\n", p, q);
}
```

```
#include <stdio.h>

void swap(int y, int z){
     int temp = y;
     y = z;
     z = temp;
}

int main() {
    int p = 42;
    int q = 24;
    swap(p, q);
    printf("%d -> %d\n", p, q);
}
```

3. Object Oriented Programming and Pass by Value/Reference

So, what does that mean for OO languages like Java and Swift? Well, if all variables are *pointers* to objects then it means that the language is inherently **pass by reference**.

3.1 What do you expect the following code snippets to print?

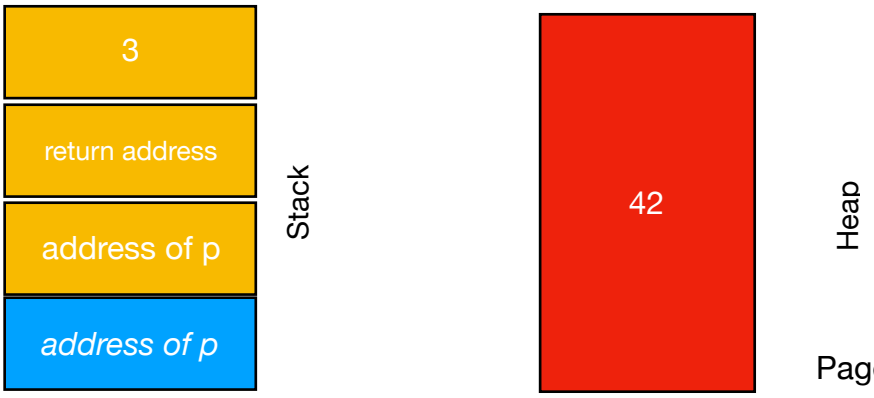
Swift

```
class Thing{
    var x: Int = 0
}

func change(t: Thing){
    t.x += 3
}

var p = Thing()
p.x = 42

change(t:p)
print(p.x)
```

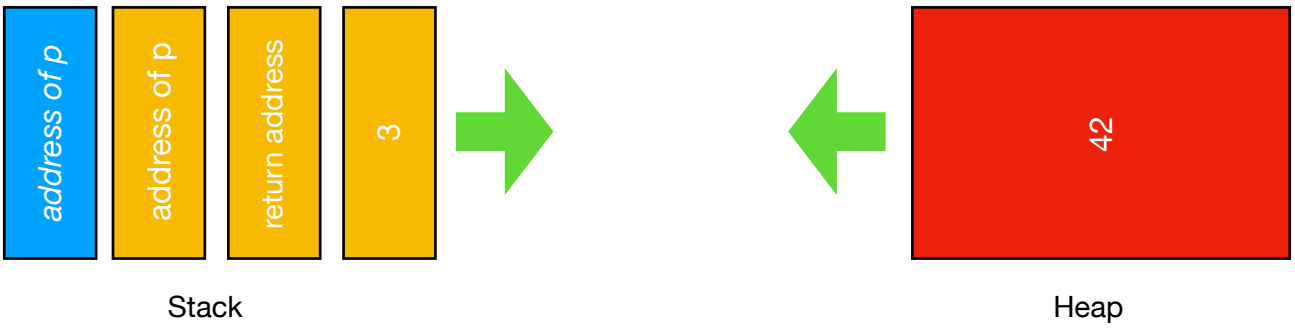
Java

```
public class Thing{
    int x = 0;

    static void change(Thing t){
        t.x += 3;
    }

    public static void main(String[] args){
        Thing t = new Thing();
        t.x = 42;
        change(t);
        System.out.println(t.x);
    }
}
```

If you answered 45 in both cases, you're right. What does this look like in memory? Well, we have to introduce the idea of a heap as well as the stack. The heap is used for dynamically sized things at compile time, and is used for objects (anytime you use new in Java/C++ or call the init function in Swift) or dynamic memory allocation (malloc in C). If we focus on the Thing class (from Swift) we get something like this:



Really this looks (something) like this in memory (if the width of the page were my entire memory space):



3.2 What would the stack look like if I have an infinitely recursive function?

4. Swift Lab Exercise

Because Swift (by default) doesn't allow modification of variables passed in as a parameter to a function, there is a slightly different method to test what's going on.

4.1 What do you expect the following code to print?

```
class Thing{
    var x: Int = 42
}

var p = Thing()
var q = p
p.x = 0
print("\(p.x), \(q.x)")
```

4.2 Do you get what you were expecting? If not, draw the call stack.

4.3 Does this tell you if you have a *reference* type or a *value* type?

4.4 Does this change if I were to use let instead of var?

4.5 Determine if the following Swift types are pass by reference or pass by value using a pattern similar to the above code snippet.

- int
- double
- float
- struct
- class
- String
- Dictionaries
- List
- Set

Compare your answers with https://developer.apple.com/swift/blog/?id=10

4.6 Does this change if I were to use let instead of var?

4.7 When might I want to deliberately use a value type over a reference type? What side effects can I avoid? What are the drawbacks?

In Swift you can declare that the parameter is **inout** which is like making it an explicit pointer in the C examples above. **Explore Swifts inout keyword with reference to the above types. What changes?**