

Passing Pointers by Value

In C, all function parameters are passed *by value*. People are frequently confused about what this means, particularly when pointers are involved. So what does “pass by value” mean?

Let’s look at an example:

```
#include <stdio.h>


void add(int *x, int *y, int *z) {
    *z = *x + *y;
}

int main() {
    int x = 1;
    int y = 2;
    int z;
    add(&x, &y, &z);
    return z;
}
```

We will walk through this program step-by-step. In the beginning, there is nothing. Note that the arrow in the following diagram points at the line *about to be evaluated*, what we call an *instruction pointer*.

```
                                #include <stdio.h>

                                void add(int *x, int *y, int *z) {
                                    *z = *x + *y;
                                }

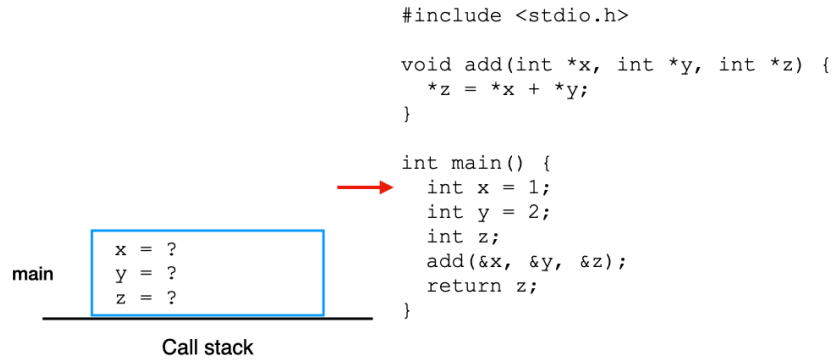
                                 int main() {
                                    int x = 1;
                                    int y = 2;
                                    int z;
                                    add(&x, &y, &z);
                                    return z;
                                }

```

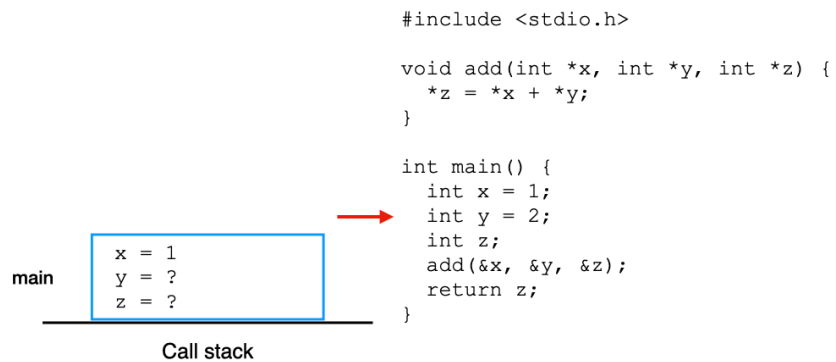
Call stack

When your computer first executes a function, the *function preamble runs*. The preamble allocates storage for the variables in the function.

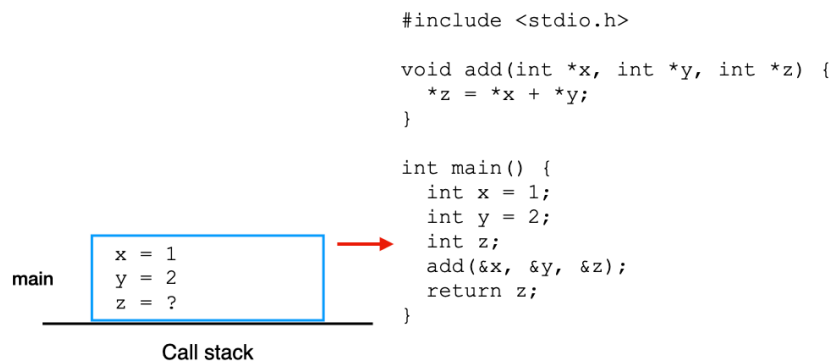
Colloquially, we sometimes refer to this as “setting up the stack frame.” If there are any function parameters, at this time, their values are copied into the local storage allocated for them. In this example, there aren’t any parameters.



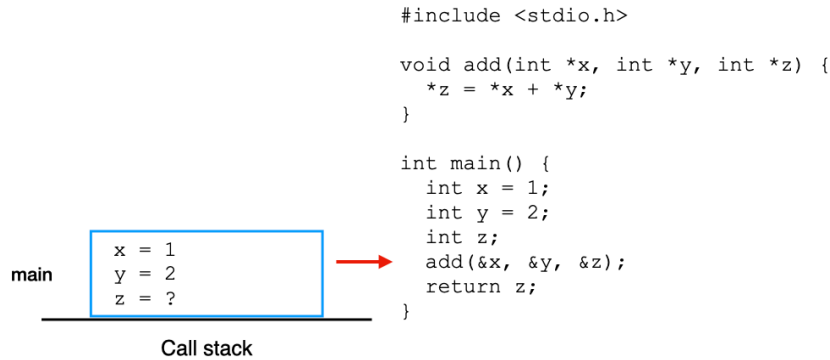
The first line of the function *assigns* the value 1 to the local variable x. What this means is that the value 1 is *copied* into the stack storage reserved for x.



The same thing happens in the next line, except that 2 is copied into the storage for y.



In the third line, we declare `z` but we don't assign anything. In C, nothing actually happens when we execute this line. The compiler already allocated local storage *before* the function ran, in the preamble. Although nothing happens when we run this line of code¹⁰, it does have to be in the program, otherwise the compiler would not have known we wanted storage for the `z` variable.



On the fourth line, the `add` function is called. But before that happens, technically, there are four more steps. Why? Because we don't know what the arguments to the function are yet. They must be *evaluated*¹¹ so that they can be *copied* into `add`'s own stack frame.

First, `add`'s function preamble is run. Do you remember the purpose of a preamble? If you don't, go back and read the earlier mention of this term. Function preambles need to be run before arguments are evaluated. Why do you think that is?

Second, `&x` is evaluated. This line "gets the address of `x`," and stores the result (an address) into the first parameter of `add`.

```

add(&x, &y, &z);
return z;

```

Third, `&y` is evaluated and the result is stored into the second parameter of `add`.

```

add(&x, &y, &z);
return z;

```

Finally, `&z` is evaluated and the result is stored into the third parameter of `add`.

```

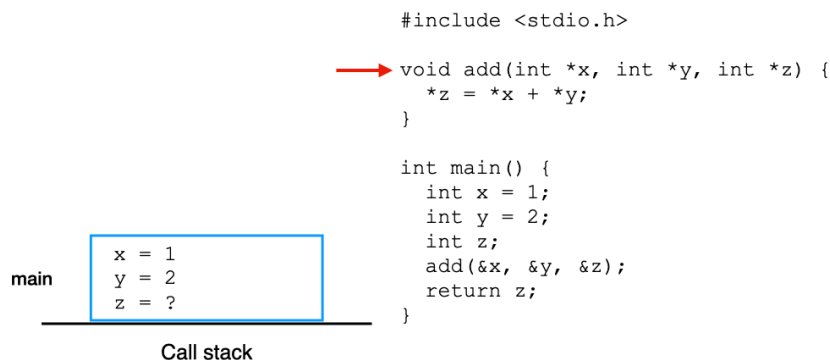
add(&x, &y, &z);
return z;

```

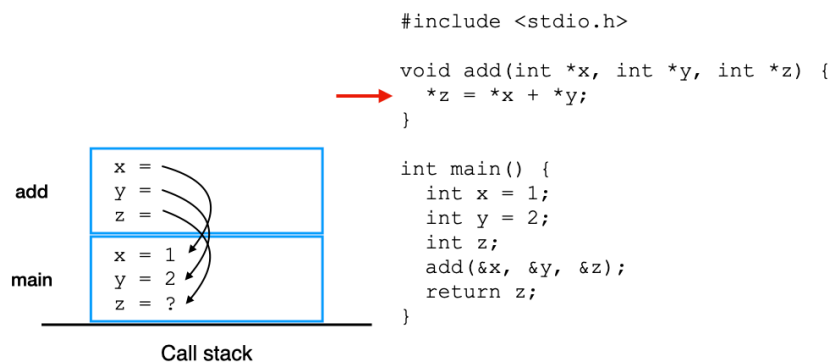
¹⁰ To be more precise, it's not so much that the computer "does nothing" when the line of code is executed; *there is no line of code* after the C program has been translated into machine language.

¹¹ There are other techniques to provide these arguments. Evaluating arguments *before* calling the function is called *eager evaluation*. Another technique, popular with functional programming languages, is called *lazy evaluation*. Lazy evaluation defers the evaluation of parameters until the moment that they are needed by the called function. In other words, lazy evaluation evaluates arguments after the function is called. The benefit of lazy evaluation is that if a function never actually uses an argument, the argument never needs to be evaluated. However, most languages choose eager evaluation because it is simple to implement.

Now that all of the arguments to `add` are evaluated, the `main` subroutine transfers control to the `add` subroutine. On most computers, this is implemented with some kind of *jump* instruction. The process starts all over again, this time for the `add` function. As before, in the beginning, there is nothing in `add`'s stack frame.



Since each argument is a pointer (they all have type `int *`), their values are *addresses*. I draw them using arrows to make things clearer, but you should know that technically, pointers are actually stored as numbers. Note that `x`, `y`, and `z` in `add` aren't just different variables than `x`, `y`, and `z` in `main`, the variables in `add` have a *different type* (`int *` as opposed to `int`). Remember that variable names are just names, and as in real life, where two *different* people can have the *same* name, two different variables in C can have the same name. We will talk about why, exactly, this duplication of names does not confuse C when we discuss *scope*. For now, observe that names are *local* to a function.



It's worth noting that `z` in `add` really does point to an undefined variable `z` in `main`. Yes, C lets you do things like that.

At this point, we execute the body of the `add` function. There's a lot going on, so as we did with function parameter evaluation, let's break the evaluation down into steps.

At a high level, we are assigning a value to a variable. What are we assigning? The result of an addition. But, since there are a bunch of `*` symbols in here, hopefully you suspect that there's more to it than that.

First, we need to get the value of the *right side* of the assignment:

$$*x + *y$$

Well, to get the value of this expression, we need to know the value of the *left side* of the addition:

$$*x$$

And to know that, we have to *dereference* `x` (that's what `*x` says in code, literally). What does it mean to dereference something? It means that we follow the pointer stored in `x` and fetch the value it points to. Now we know `*x`. It's 1.

$$1 + *y$$

What do you think we do next? We need to know the *right side* of the addition. `*y` does essentially the same thing as `*x`. When we dereference `y`, we follow its pointer and find 2.

$$1 + 2$$

The *right side of the assignment* can now be evaluated since we have all of the values. $1 + 2 = 3$.

$$*z = 3$$

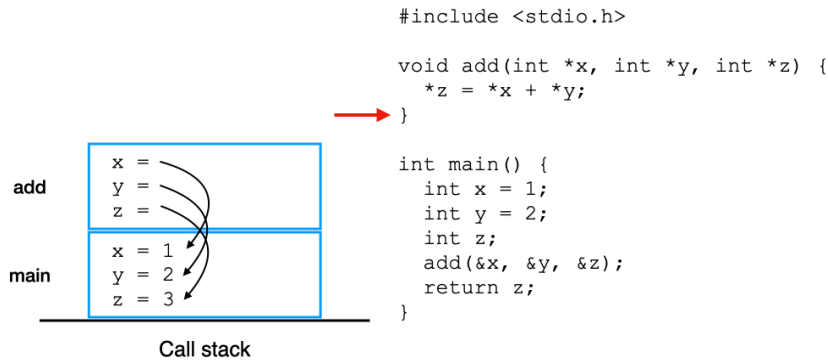
But wait... we still have a `*` on the *left side of the assignment*. That's because `z` is also a pointer. We don't actually want to store 3 in `z`. That wouldn't make sense because 3 is an `int` while `z` is a pointer. Instead, we want to *follow* the pointer stored in `add`'s `z` and store the value in this other location. That other location happens to be `z` in `main`.

When you see a pointer on the left side of an assignment, what happens is the following:

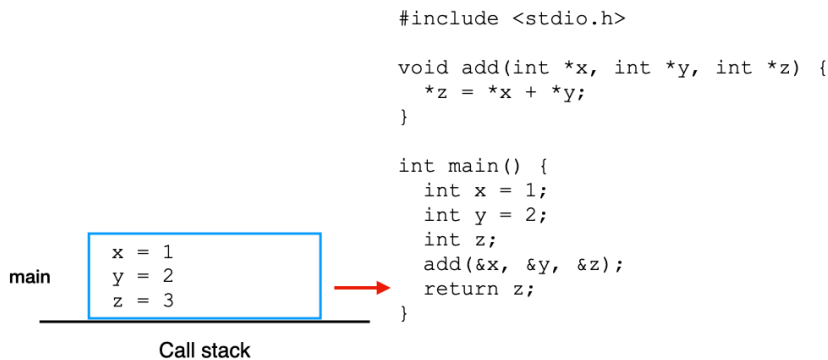
1. Evaluate the right hand side (we did this already).
2. Store the value of the right hand side in the location *pointed to* by the left hand side.

So what happens now is that we store 3 in `z` in `main`.

Note that this is why `add` can get away with returning `void`. `add` directly manipulates memory stored in `main`'s stack frame. Also notice that we did all this cool pointer stuff without any mention of `malloc` (i.e., without variables having *allocated storage duration*). Pointers and storage duration are different, but complementary concepts, as will become clear in the next step.

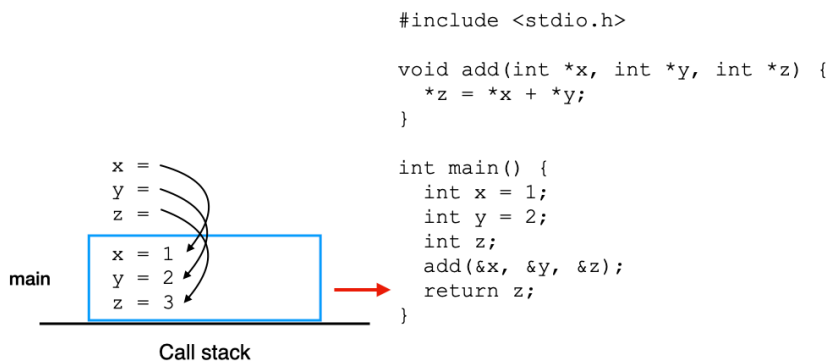


Now that we're at the end of the `add` function, `add`'s *function epilogue* runs. It says how to restore the stack to its state before `add` was called. The storage for `x`, `y`, and `z` in `add` goes away. The storage for each variable goes away because each was allocated with *automatic storage duration*—in other words, it goes away automatically because we *did not* use `malloc`.



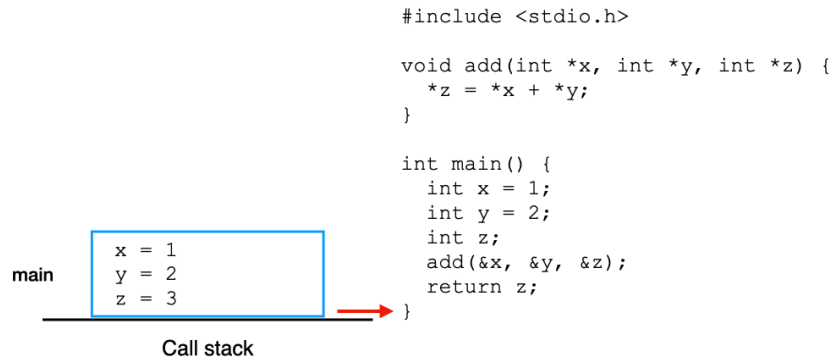
It is worth mentioning that the dirty little secret in most C implementations is that those values don't *really* go away. The values are technically still stored in those locations. If you are clever, you can still even read them¹². What you should not do, however, is count on those values staying there, because the moment you call another function, they'll probably be overwritten.

¹² And this is something that clever hackers may do!

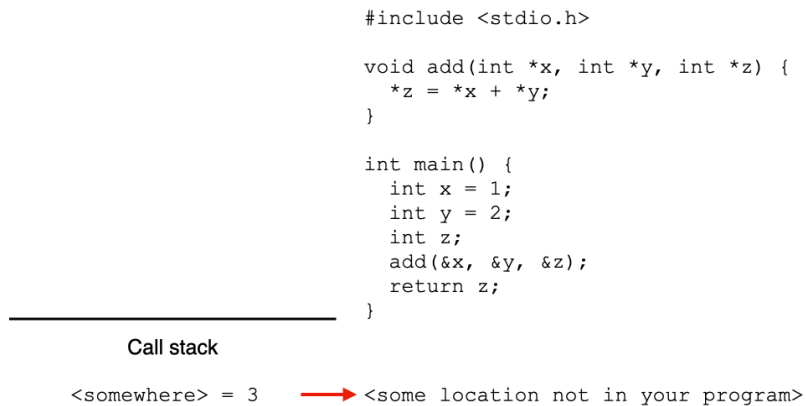


Now that we're back in `main`, we pick up where we left off and finally return `z`. Ever wondered what return `z` actually means? It means "copy the value stored in `z` into the location specified by the calling function and then jump to the function epilogue." So we copy 3¹³ and move to the epilogue.

¹³ Where, exactly, 3 gets copied depends on which function called `main`, which is a detail I've omitted here.



Finally the program is done; `main`'s epilogue is run and the final stack frame is torn down.



At this point, what to do next is somebody else's problem.