

# Unit 13: Call Stack

In this unit, we will delve deeper into how variables are stored in the memory and what happens when we call a function.

## Learning Objectives

After this unit, students should:

- understand the purpose of the call stack
- be aware that the stack frame represents the memory allocated to a particular function call
- be aware that the stack frame for a function is deallocated once the function terminates
- understand what occurs in a stack frame when a variable is declared
- understand what occurs in a stack frame when a variable is assigned
- understand how function parameters are handled within the stack frame
- be able to trace the contents of the call stack through nested function calls
- be aware that there are three different types of variables: automatic, global, and static
- be aware that the operating system limits the size of the call stack for each program

## Stack Frame

Modern OS typically divides the memory into several regions. The region that we are concerned with for now is called the *call stack*.

Every function invocation causes the OS to allocate some memory on the call stack to store (among other things) the parameters passed into the function and the variables declared and used in the function. The memory allocated to each function call is called a *stack frame*. When a function returns, the stack frame is deallocated and freed up for other uses.

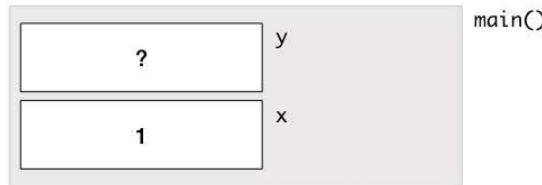
Let's start with the following simple program as an example.

```
1 int main()
2 {
3     long x = 1;
```

```

4     long y;
5 }
```

When the OS runs the program above, it invokes, or calls, the function `main`. A new stack frame is created for `main()`. There are two variables `x` and `y` declared in `main`. Recall from [Unit 2](#) that a variable is a location in the memory which holds a value. Thus, the stack frame of the `main` will include these two variables. We initialize `x` to 1 in the code above, so the value 1 will be placed in the memory location of `x`. The variable `y` remains uninitialized, so it will contain whatever value that happens to be in the memory at that time.



Now, let's consider the program<sup>1</sup>:

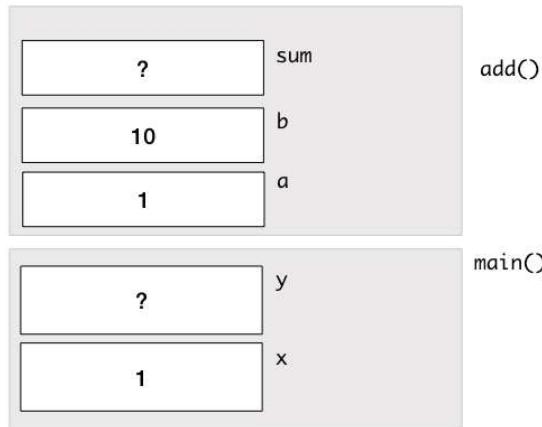
```

1 long add(long a, long b) {
2     long sum;
3     sum = a + b;
4     return sum;
5 }
6
7 int main()
8 {
9     long x = 1;
10    long y;
11    y = add(x, 10);
12 }
```

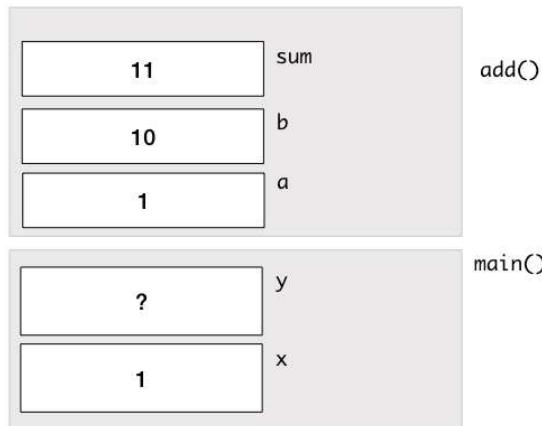
Now, the program invokes the function `add` with two parameters, using `x` and 10 as arguments. What the OS does when `add` is called, is that it allocates another stack frame for `add`. We can visualize the stack frame for `add` as being placed on top of the stack frame of `main`. The stack frame for `add` includes two variables corresponding to the parameters `a`, `b`, and the variable `sum` declared in the function `add`.

When the stack frame for `add` is created, `sum` is uninitialized, but `a` is initialized to whatever the value of `x` is when the function is invoked (1 in this example), and `b` is

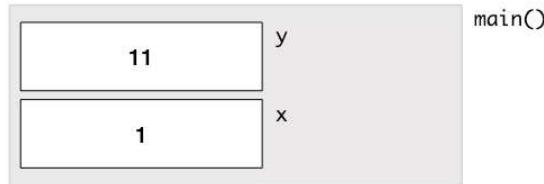
initialized to 10, since that is the argument passed into `add`.



After the stack frame for `add` is set up, the code is executed. The memory location for `sum` is then initialized to the sum of `a` and `b` (11 in this example), and the return statement is executed.



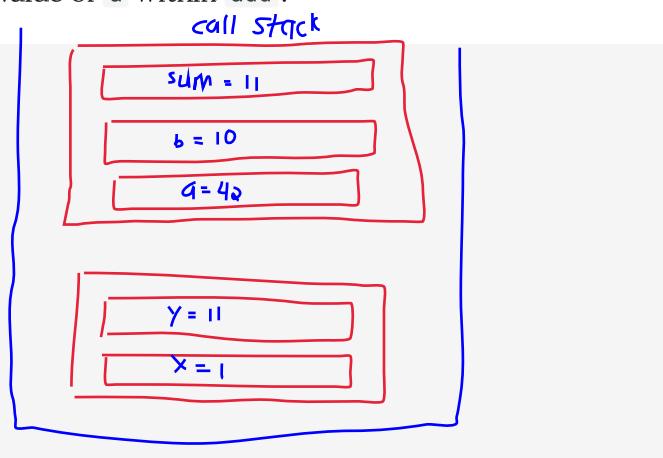
When a function returns, the stack frame for `sum` is removed. The variables `sum`, `a`, `b` crumble into dust and no longer exist in the memory. The value of the variable being returned (`sum` in this case) is then copied back to the stack frame of the `main` (the `caller`) In this example, this value is copied into the memory location of `y`.



What would happen if we change the value of `a` within `add`?

```

1 long add(long a, long b) {
2     sum = a + b;
3     a = 42; // change a
4     return sum;
5 }
6
7 int main()
8 {
9     long x = 1;
10    long y;
11    y = add(x, 10);
12 }
```

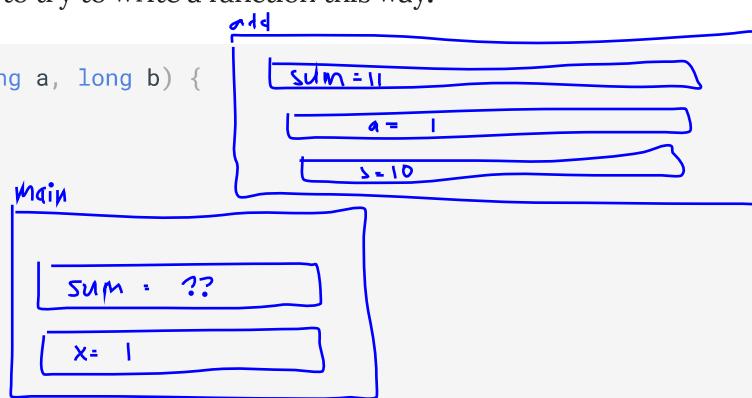


In this example, we change the value of `a` in the call stack of `add` to 42. This does not affect any variable in `main`, since the changes apply only within the stack frame of `add`.

A common bug by beginners is to try to write a function this way:

```

1 void add(long sum, long a, long b) {
2     sum = a + b;
3 }
4
5 int main()
6 {
7     long x = 1;
8     long sum;
9     add(sum, x, 10);
10 }
```



Here, there are two versions of `sum`, one in the stack frame of `main`, the other in the stack frame of `add`. When we change `sum` in `add`, we are changing a local copy which will eventually get deleted. The assignment to `sum` in Line 2 does not affect the variable `sum` on Line 8 of `main`.

This explanation explains why a function is a black box, we can only pass in parameters and get a value in return. Nothing else gets in and out. There are no "side effects" -- whatever happens in the function, stays in the function and does not affect the caller (excepts if it reads or writes to the standard input/output). Such a function is sometimes called a "pure function".

## Automatic Variable

Because of the way the memory for the variables are automatically allocated and deallocated when a function is invoked and when the function returns, we call these variables *automatic variables*.

There are two other types of variables, *global variable*, which we have mentioned is a very very bad programming practice in Lecture 5 and is banned from CS1010, and *static variable*, which could be useful in the rare occasion but we shouldn't concern ourselves with this type of variable in CS1010 for the moment.

## Stack Size

The OS typically reserves a limited amount of memory for each program for the stack. You can find out, on your system, what is the maximum allocated stack size with the command

```
1 | ulimit -s
```

On our CS1010 PE hosts, the limit is 8 MB.

## Problem Set 13

### Problem 13.1

Trace through what gets stored in the call stack when we run the following programs:

```
1 | #include <math.h>
2 |
3 | long square(long x)
4 | {
5 |     return x*x;
6 | }
7 |
8 | double hypotenuse_of(long base, long height)
9 | {
10|     return sqrt(square(base) + square(height));
11| }
```

```

13 int main()
14 {
15     hypotenuse_of(3, 4);
16 }
```

## Problem 13.2

Trace through what gets stored in the call stack when we run the following programs:

```

1 #include "cs1010.h"
2
3 long factorial(long n)
4 {
5     if (n == 0) {
6         return 1;
7     }
8     return factorial(n-1) * n;
9 }
10
11 int main()
12 {
13     factorial(3);
14 }
```

## Problem 13.3

What will be printed by the program below? Trace through what gets stored in the call stack when we run the following programs:

```

1 #include "cs1010.h"
2
3 void incr(long x) {
4     x += 1;
5 }
6
7 int main()
8 {
9     long x = 10;
10    incr(x);
11    incr(x);
12    cs1010_print_long(x);
13 }
```

- 
1. The variable `sum` is not necessary and the function can be better written as a one-liner `return a + b;`. But I introduce the additional variable `sum` here to make the example more meaningful in explaining about stack frame. 