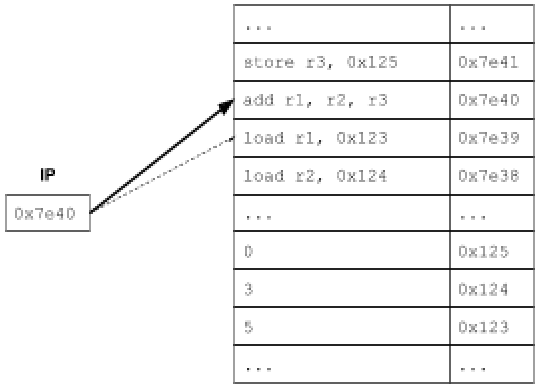
Function Calls

Kevin Burleigh, 2020-10-17

How A Program Runs

A program is loaded into memory. Every instruction has an address.
The instruction pointer (IP) is the location of the next instruction to be executed.
Instructions are loaded and the IP is incremented until the program exits.



Functions

A **function** is a **named block** of **reusable instructions**.

Sometimes programs will use **lambdas**, which are essentially **anonymous functions**.

Anatomy of a Function

Functions have four main parts:

1. a return type
2. a function name
3. parameters
4. a body of instructions

```
RETURN_TYPE FUNC_NAME(PARAM1,PARAM2,...) {
    BODY;
}
```

Functions are called by passing in an **argument** for each **parameter** and assigning the returned value to a local variable:

```
int y = func(a,7);
```

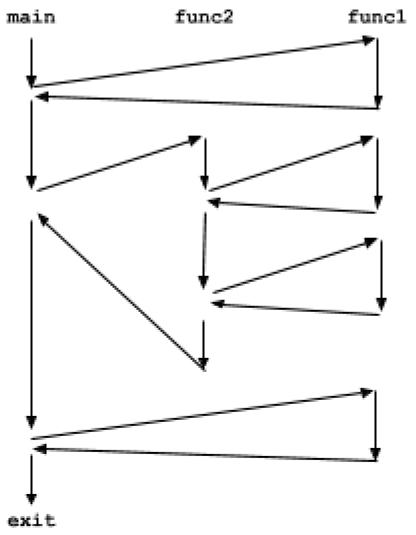
Notice that arguments can be either explicit values or variables.

Flow of Control

Suppose we have the following code:

```
void func1() {
}

void func2() {
    func1();
    func1();
}

void main()
{
    func1();
    func2();
    func1();
}
```

The **flow of control** maps to the following graph:



The computer needs some way to keep track of:

- which code is currently executing
- which variables are currently **in scope**
- which code should run next when a function returns

It turns out this can all be done with a **call stack**.

The Call Stack

For our purposes here, we can envision the call stack storing the following for each function call:

- parameters
- local variables

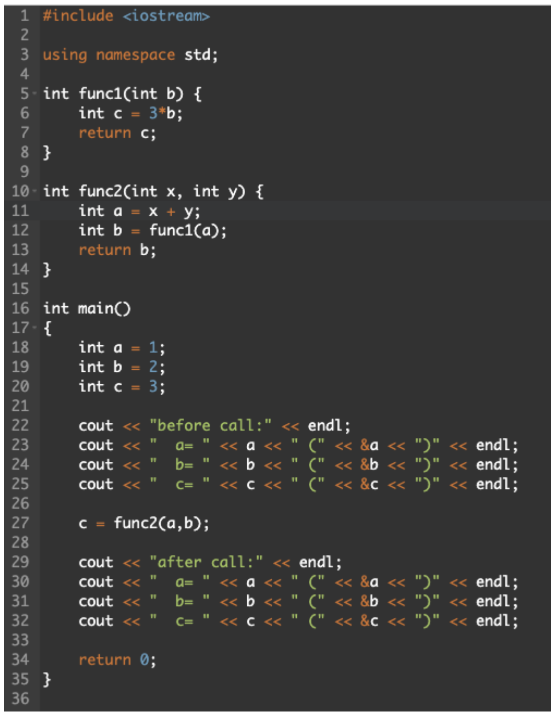
- return value destination
- instruction pointer

When a function is called, the following sequence takes place:

- a new **stack frame** is created on the call stack
- its parameters are initialized
- execution begins
- the return value is placed in its final destination
- the stack frame is destroyed
- the calling function continues where it left off

Example

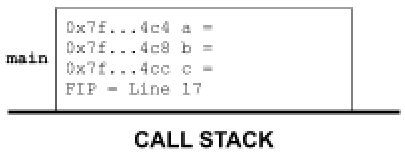
We’re going to run through the following code, keeping careful track of what is happening on the call stack:



```
1  #include <iostream>
2
3  using namespace std;
4
5  int func1(int b) {
6      int c = 3*b;
7      return c;
8  }
9
10 int func2(int x, int y) {
11     int a = x + y;
12     int b = func1(a);
13     return b;
14 }
15
16 int main()
17 {
18     int a = 1;
19     int b = 2;
20     int c = 3;
21
22     cout << "before call:" << endl;
23     cout << "  a= " << a << " (" << &a << ")" << endl;
24     cout << "  b= " << b << " (" << &b << ")" << endl;
25     cout << "  c= " << c << " (" << &c << ")" << endl;
26
27     c = func2(a,b);
28
29     cout << "after call:" << endl;
30     cout << "  a= " << a << " (" << &a << ")" << endl;
31     cout << "  b= " << b << " (" << &b << ")" << endl;
32     cout << "  c= " << c << " (" << &c << ")" << endl;
33
34     return 0;
35 }
36
```

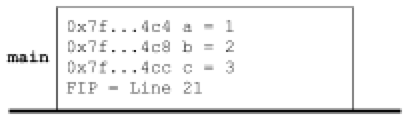
Initially, the call stack is empty:

The program begins by calling the `main` function. This pushes a new stack frame onto the call stack:



Notice that `main` has no parameters, and that its local variables (`a`, `b`, and `c`) are currently uninitialized. (Technically, for modern C/C++ variants, these will be initialized to zero, but since we don't explicitly set them, we'll just assume we can't trust their values.)

Execution is currently on Line 17 of the source file. We advance to Line 21, and now the local variables are initialized:

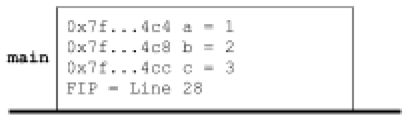


CALL STACK

We've reached the print statements, which output something like this:

```
before call:
  a= 1 (0x7ffd5f82ea44)
  b= 2 (0x7ffd5f82ea48)
  c= 3 (0x7ffd5f82ea4c)
```

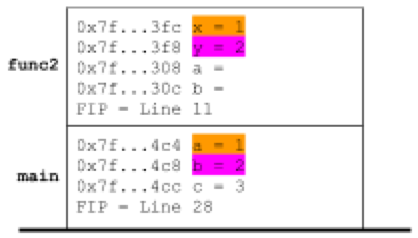
Once we reach the call to `func2`, the stack looks like this:



CALL STACK

Note that `main`'s function's instruction pointer (FIP) is pointing to the instruction to be executed after we return from the call to `func2`.

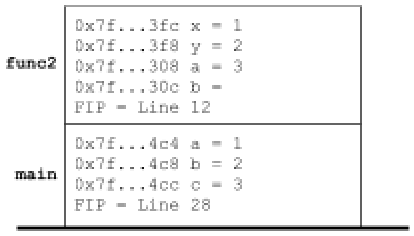
Calling `func2` pushes a new stack frame. The function's parameters, `x` and `y`, are initialized to the arguments, `a` and `b`, provided by the caller:



CALL STACK

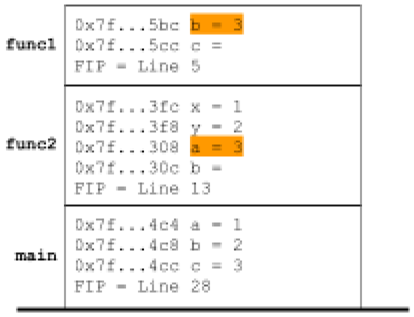
Notice that there are two different `a`s and two different `b`s, one each in `main` and `func2`. (We can tell they are different because they have different memory addresses.) This is not a problem, though, since we know what function we're currently executing, and we can easily find the appropriate variable using its stack frame.

We can now start executing `func2`:



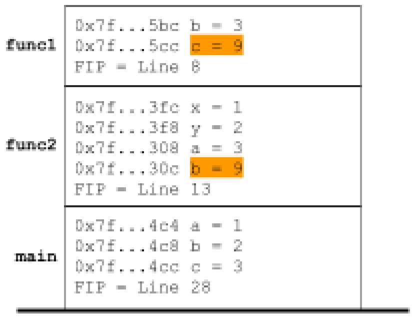
CALL STACK

The local variable `a` is now initialized, and we're about to call `func1`. This pushes another stack frame onto the call stack:



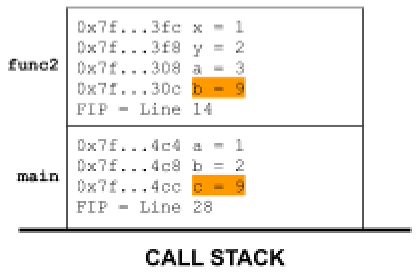
CALL STACK

As before, the parameter `b` is initialized to the argument provided by the caller. Just before we return from `func1`, `c` has been initialized and we've copied it into the return variable `b` in `func2`:

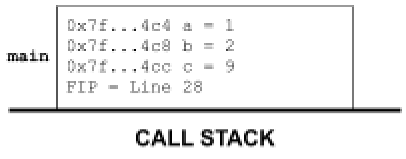


CALL STACK

The stack frame for `func1` is destroyed, and we pick up execution on Line 13 inside `func2`. Just before `func2` returns, we have the following on the stack:



The return value, `b`, has been copied into the local variable `c` inside main. We destroy the stack frame for `func2` and continue executing `main`:



We reach the second set of print statements and get the following:

```
after call:
  a= 1 (0x7ffd5f82ea44)
  b= 2 (0x7ffd5f82ea48)
  c= 9 (0x7ffd5f82ea4c)
```

We then reach the end of `main`, and it exits:

Now that there is nothing left on the call stack, the program ends.

Passing Arguments as Parameters

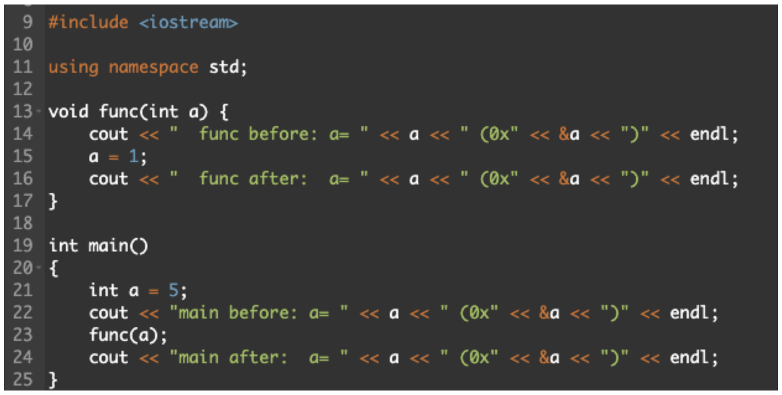
There are two ways to set a function's parameters:

- by value
- by reference

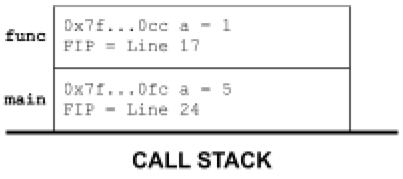
Knowing the difference is essential to understanding and explaining program behavior.

Pass By Value

When an argument is **passed by value**, it is copied. We can see this in the following code:


```
 9 #include <iostream>
10 
11 using namespace std;
12 
13 void func(int a) {
14     cout << "  func before: a= " << a << " (0x" << &a << ")" << endl;
15     a = 1;
16     cout << "  func after:  a= " << a << " (0x" << &a << ")" << endl;
17 }
18 
19 int main()
20 {
21     int a = 5;
22     cout << "main before: a= " << a << " (0x" << &a << ")" << endl;
23     func(a);
24     cout << "main after:  a= " << a << " (0x" << &a << ")" << endl;
25 }
```
which outputs:

```
main before: a= 5 (0x0x7ffdfb8300fc)
  func before: a= 5 (0x0x7ffdfb8300cc)
  func after:  a= 1 (0x0x7ffdfb8300cc)
main after:  a= 5 (0x0x7ffdfb8300fc)
```
We can see why by tracking the stack frames during execution. Just before `func` returns, the call stack is in the following state:

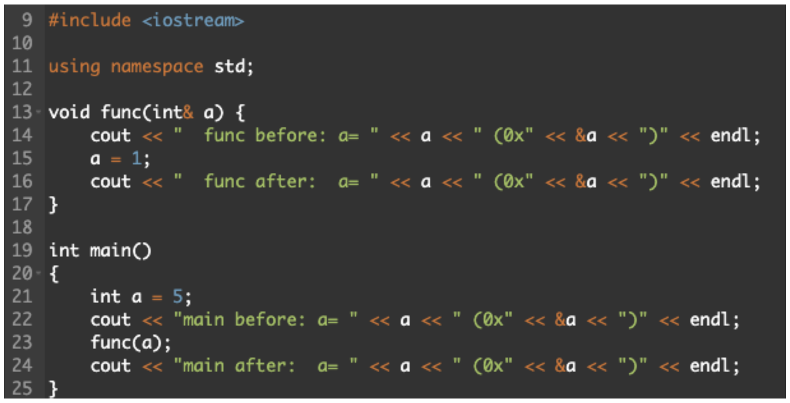


We can see that the value of the local variable `a` inside `func` is actually a copy of the value of the local variable `a` inside `main` (note that it has a different address). So changing its value has no effect outside of `func`, as reflected in the program's output.

Pass By Reference

When an argument is **passed by reference**, the parameter becomes an alias for something that already exists - nothing is copied.

Consider the same program as above, but with one tiny change:


```
 9 #include <iostream>
10 
11 using namespace std;
12 
13 void func(int& a) {
14     cout << "  func before: a= " << a << " (0x" << &a << ")" << endl;
15     a = 1;
16     cout << "  func after:  a= " << a << " (0x" << &a << ")" << endl;
17 }
18 
19 int main()
20 {
21     int a = 5;
22     cout << "main before: a= " << a << " (0x" << &a << ")" << endl;
23     func(a);
24     cout << "main after:  a= " << a << " (0x" << &a << ")" << endl;
25 }
```

Note that we now have:

```
void func(int& a) { ... }
```

instead of:

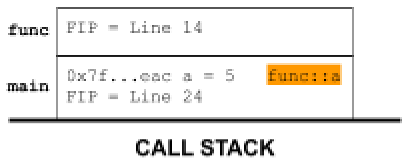
```
void func(int a) {...}
```

The ampersand (&) on the function parameter type means it should be passed by reference (alias), not by value (copy).

This program outputs:

```
main before: a= 5 (0x0x7fff053a0eac)
  func before: a= 5 (0x0x7fff053a0eac)
  func after:  a= 1 (0x0x7fff053a0eac)
main after:  a= 1 (0x0x7fff053a0eac)
```

We can see why by examining the call stack. Just before `func` begins executing, we have the following:



Notice that there is no local variable `a` in `func`'s stack frame. That's because it's actually an alias for the variable `a` inside `main`'s stack frame, as highlighted above. When we change its value, we're actually changing the value in `main`. This is reflected in the program's output. Notice that all of the `a`s actually have the exact same address - they're one and the same!

The Heap

Unlike the contents of stack memory, which are automatically created or destroyed during function calls and returns, items in heap memory can exist across many function calls. These items need to be managed by (a) the developer, or (b) by the language garbage collector.

Passing Handles By Value

In most higher-level languages (ruby, python, java), variables are (or at least appear to be for all practical purposes) handles to objects on the heap. These handles are stored on the stack, and passed by value to functions.

Consider the following python code:

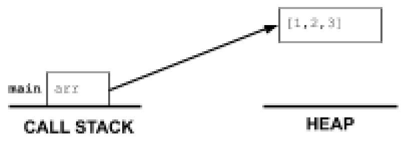
```
def clear_array(x):
    x = []

def main():
    arr = [1,2,3]
    print(arr)
    clear_array(arr)
    print(arr)

main()
```

which prints out:

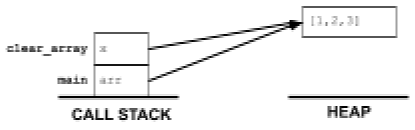
```
[1, 2, 3]
[1, 2, 3]
```

Why didn't the `clear_array` function actually clear the array? Analyzing the call stack reveals the answer.

When `main` is called, we create a new stack frame. That stack frame contains the local handle object `arr`, which points to an array on the heap after the first line is run:



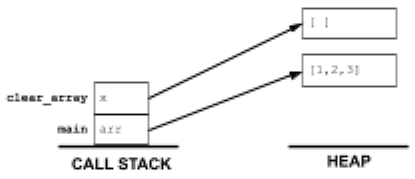
When the `clear_array` function is called, we create another stack frame and copy the handle object `arr` into `x`:



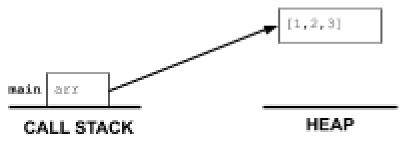
The line:

```
x = []
```

creates a new, empty array on the heap and points `x` to it. Note that this has no effect on `arr`, because `x` is a copy of `arr`:



When `clear_array` returns, its stack frame is destroyed:



and the second print statement prints out `arr` for a second time.