MTH 4300, Lecture 5

Memory; The Stack and Stack Frames; Reference Variables; Reference Parameters

E Fink

February 18, 2025

1. Memory

An object is a FIXED parcel of memory with associated a FIXED data type. Let's explore this more. Consider a program whose `main()` function is

```
int main(){
      int x = 5, y = 7;
      y = x;
}
```

The variables `x` and `y` will be associated with a particular 4 bytes of memory (since `int`s are stored in 4 bytes $= 32$ bits, usually), each of which are located at particular addresses. During program execution, `x` and `y` will always refer to those exact memory locations.

So, when we write `y = x;`, the value contained in the memory space for `x` is copied into the memory space for `y`, writing over what was present before. This behavior is referred to as *value semantics*: when an assignment occurs, the assigned-to variable refers to the same object, but that object's value is changed.

The alternative to value semantics is *reference semantics*: during an assignment, the assigned-to variable receives the *address* of the object on the right side. (Python uses reference semantics, whether you noticed it or not!)

Memory

We can explore this with the *address* operator. If `x` is a variable, then `&x` evaluates to the memory location of that variable. You can print these addresses, although it will be shown in hexadecimal – this is base 16, which has 16 digits (0-9, a = ten, b = eleven, c = twelve, d = thirteen, e = fourteen, f = fifteen), and place values which are powers of 16 – shown with a prefix of `0x`.

(For example, `0x5ac` would represent the base-10 number $5 \times 16^2 + 10 \times 16^1 + 12 \times 16^0 = 5(256) + 10(16) + 12(1) = 1452$.)

L5x1_address.cpp

An important point about arrays: they are stored in consecutive memory addresses. In the last examples, note that the addresses of `x[0]`, `x[1]`, `x[2]`, `x[3]`, all differ by 4 – since `ints` are stored in 4 bits.

Also, printing out the name of an array usually leads to the address of its first element printing out – unless you happen to have an array of `chars`, in which case you should avoid printing out the name, unless your array ends with the `'\0'` character.

2. The Stack and Stack Frames

Whenever a program is run, the memory space reserved for that program is generally divided into three pools: the *static memory*, *stack memory*, and *heap memory*. The *stack* is the first one worth understanding: it is where local variables are stored – most of the memory we've consciously utilized so far.

Every time a function is called in the execution of your program, a *stack frame* is added to the stack*. What is a stack frame? It is a record which reserves all the memory needed for an individual use of a function. Memory to store all the function's local variables, as well as the return address, is reserved (perhaps along with other data that we won't concern ourselves with). When the function's execution is finished, the stack frame is removed from the stack, and all the function's local variables go out of scope.

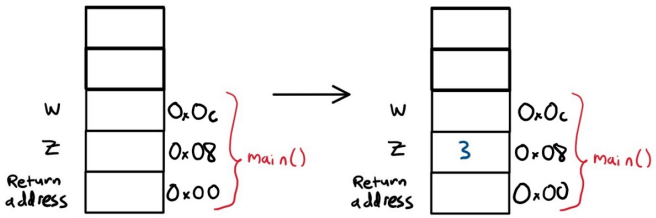
(*Actually, how function calls are executed is compiler-dependent, but what we describe here is both very common, and an excellent way to understand how C++ behaves in general.)

The Stack and Stack Frames

Consider the following program (**L5x2_square.cpp**):

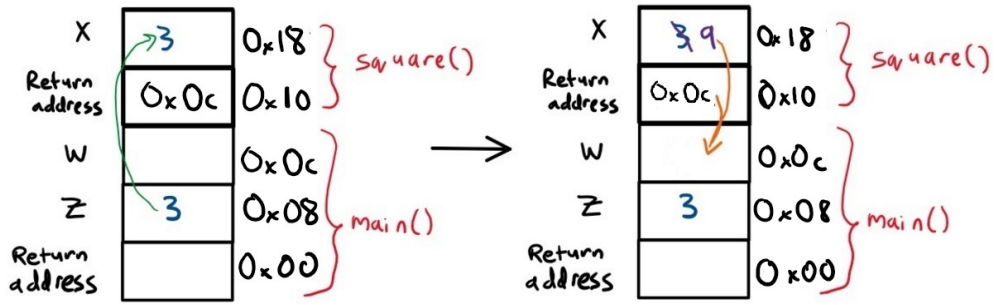
```
int square(int x) {
   x = x*x;
   return x;}
int main() {
   int z = 3;
   int w = square(z);
   cout << w;}
```

Program execution always starts with a call to `main()`. Two 4-byte regions are reserved in the stack: ones for the variables `z` and `w`, as well as a space for the address where the return value goes. Then, `main()` starts executing, and `z` gets supplied with the value 3. Below is a simplified illustration:

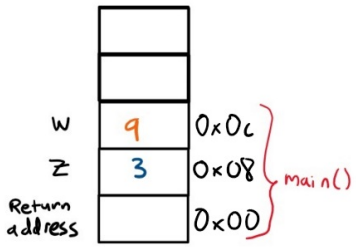


The Stack and Stack Frames

Then, the `square()` function is called: this causes a new stack frame to be placed on the call stack. This frame will have memory reserved for the local variable `x`, as well as for the address of the return value (and some other things we'll ignore). The value of `z`, which is 3, will get copied into the memory space for `x`; and the return address of the call will be the address of `w`.



Then, the body of `square()` will execute. First, the value of `x * x` will be computed, and then the number 9 will be written into the address for `x`. Next comes the return statement, which will simply copy the value 9 to where ever the return address says to put it.

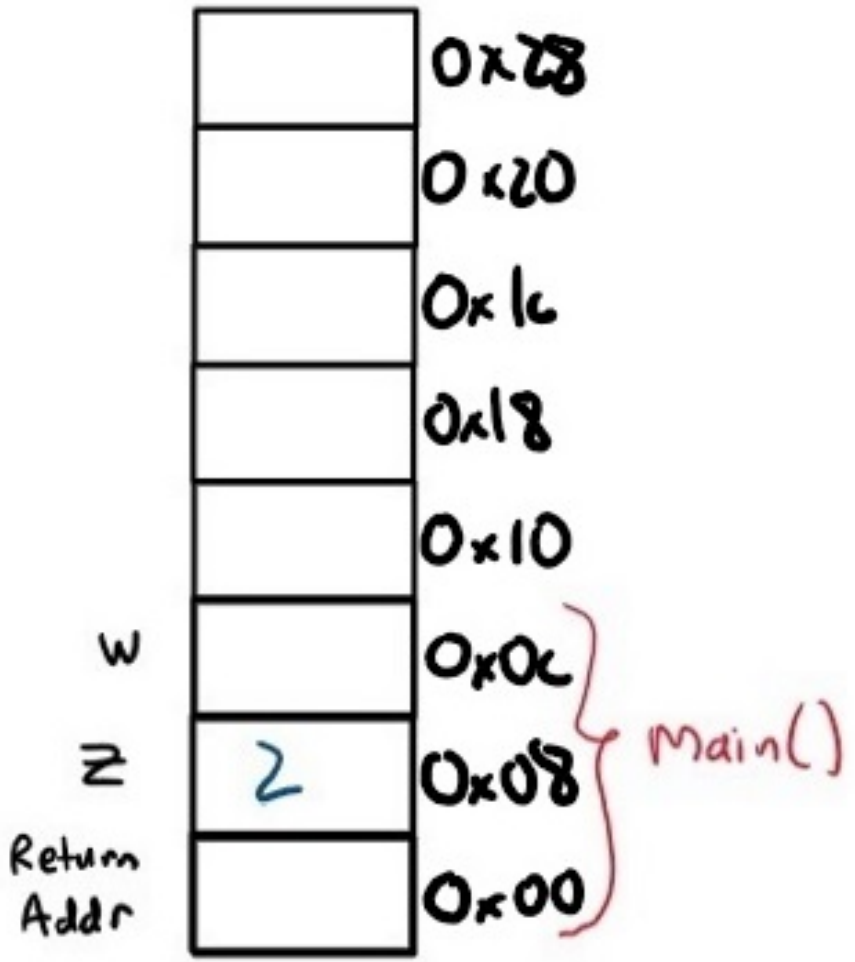


Now, the top stack frame is removed from the stack. Execution of `main()` now continues.

The Stack and Stack Frames

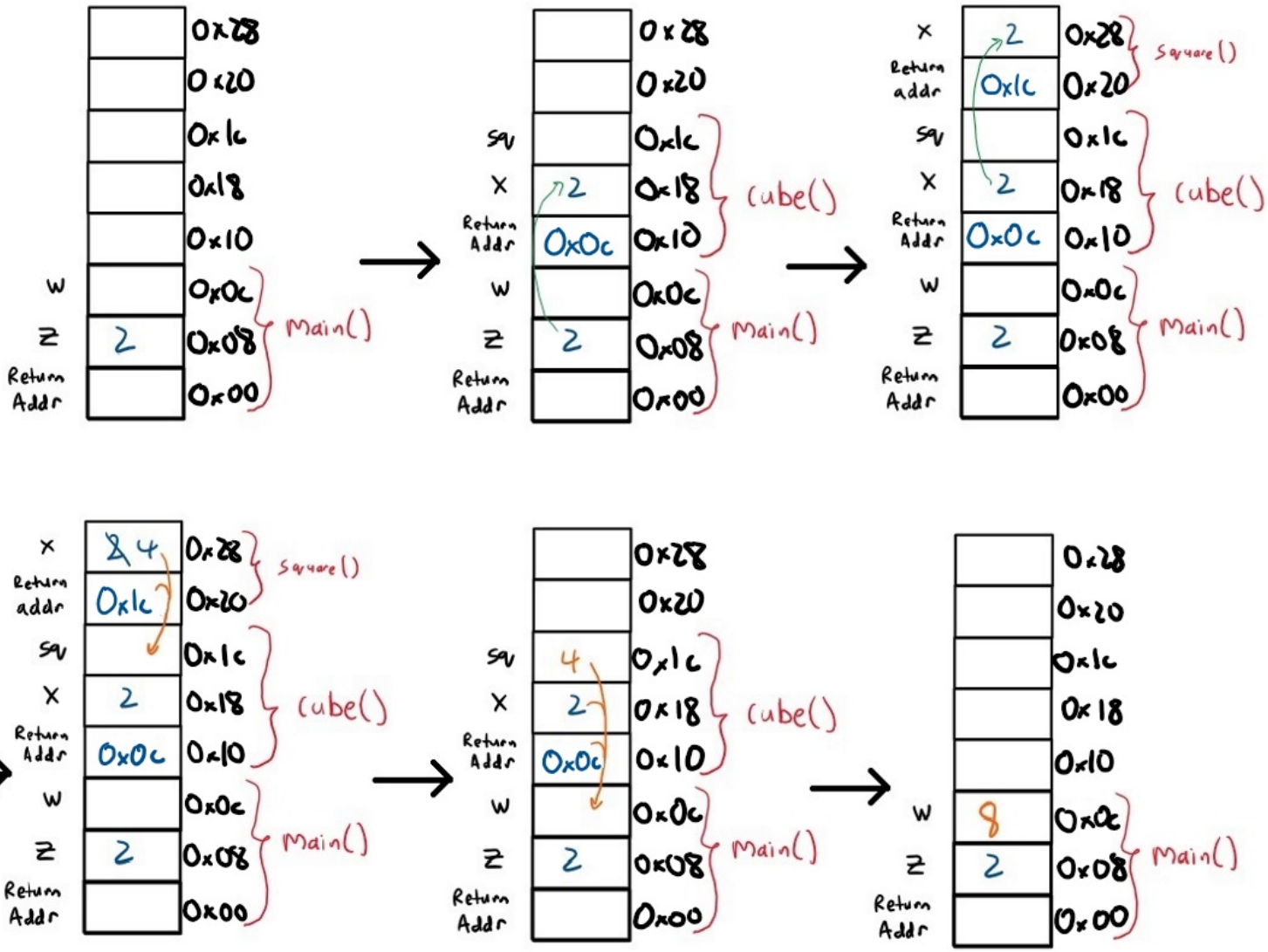
Now, consider the following program (**L5x3_cube.cpp**):

```
int square(int x) {
   return x*x;}

int cube(int x) {
   int sq = square(x);
   return sq * x;}

int main() {
   int z = 2;
   int w = cube(z);}
```



The Stack and Stack Frames

Here's a cartoon depicting what takes place on the last slide. Note that these pictures make clear the difference between the two variables named `x`, one which is local to `cube()` and one which is local to `square()`.



Warning: we are about to introduce a second use for the ampersand (&) operator, which is clearly related, but DIFFERENT from the one we saw before.

C++ also has its ways of supporting reference semantics. One way is by the use of *reference variables*. Reference variables are declared like normal variables, except with an ampersand preceding their name – and they have to be *immediately initialized to some other variable of the same type*.

For example, in the code in the example, the line
`int &myref = x;`
does not allocate any new space to `myref`; in fact, `myref` will refer to the same address as `x`. You can then use `myref` exactly like any other variable – but be aware that changes to `myref` will also change the value of `x`, and vice-versa.

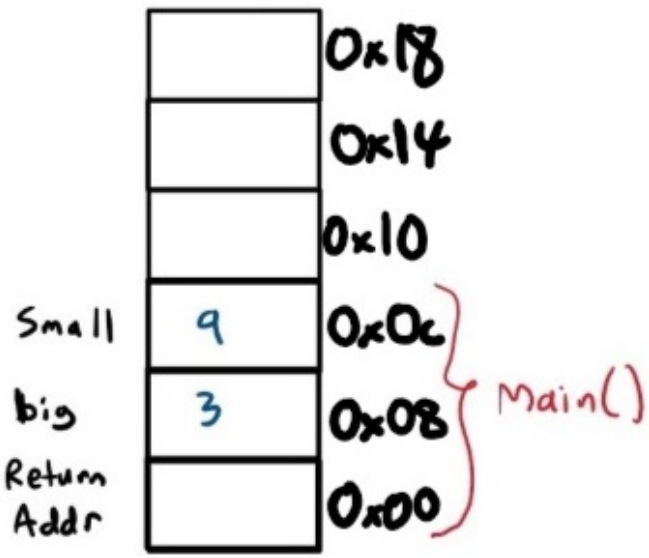
L5x4_ref.cpp

(Is this useful? It will be when we use it with functions.)

4. Reference Parameters

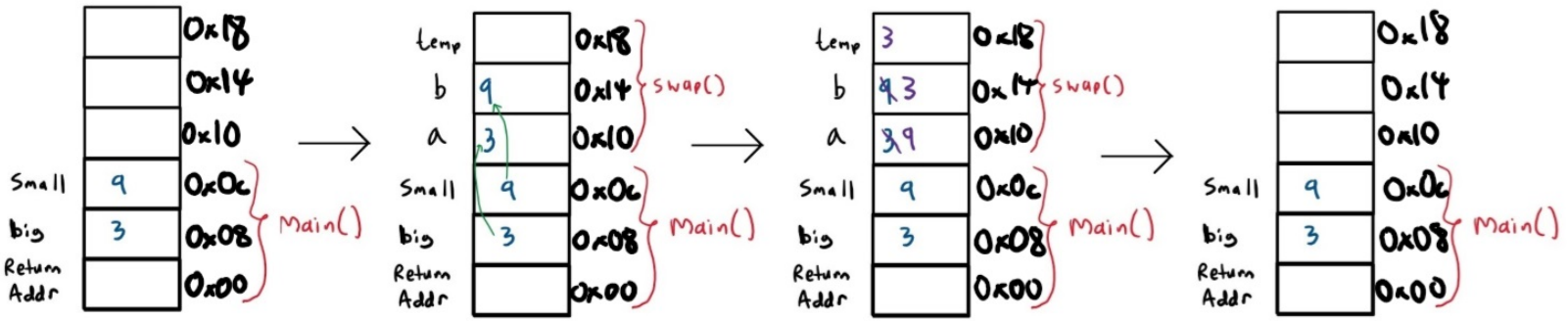
Consider this code, which has a neat idea for swapping variables, but doesn't work (see **L5x5_swap.cpp**):

```
void swap(int a, int b){
   int temp = a;
   a = b;
   b = temp;}

int main() {
   int big, small;
   cin >> big >> small;
   if (big < small) {
      swap(big, small);
   }
   cout << "big is " << big << " & small is " << small;}
```



Reference Parameters

This would be the corresponding cartoon depicting the stack frame. You can see why `big` and `small` don't get changed: while `a` and `b` get swapped, they are definitely distinct objects from `big` and `small`, which stay the same.



Reference Parameters

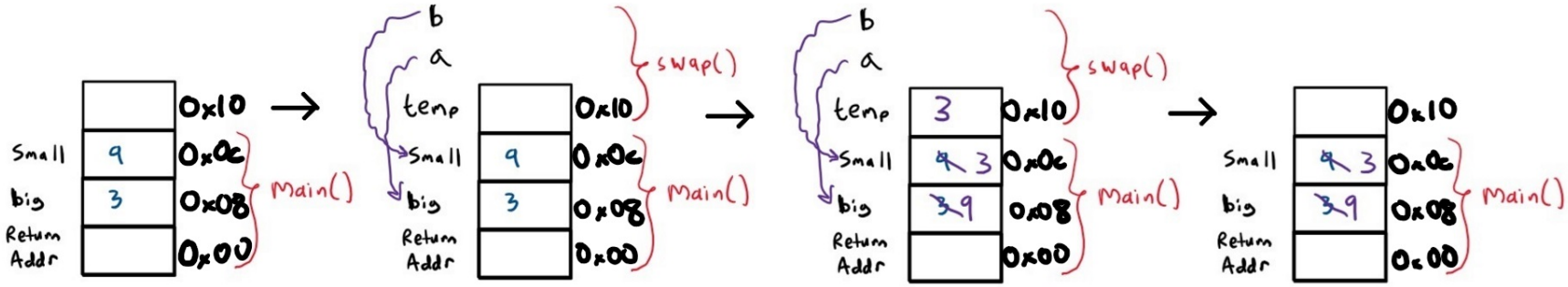
Now, consider this alternative: the parameters `a` and `b` are *reference* variables. Now, instead of `a` and `b` being new objects in memory, they are simply references to the objects stored by `big` and `small`.

As a consequence: `swap()` directly manipulates non-local variables!

```
void swap(int &a, int &b){ // THE ONLY CHANGES!!!
   int temp = a;
   a = b;
   b = temp;}

int main() {
   int big, small;
   cin >> big >> small;
   if (big < small) {
      swap(big, small);
   }
   cout << "big is " << big << " & small is " << small;}
```

Reference Parameters

Here is the cartoon of the last slide's code executing. a and b won't have their own boxes; instead, any reference to them will just be redirected to other objects.



We've just illustrated one use of reference parameters – writing functions that can directly manipulate the variables which are passed into them.

Another major advantage of reference parameters is visible in the cartoons. If you look back a couple of slides, you will stacks where the same value has been copied from one function's frame to another's. The act of copying a value from one memory location to another takes time – and if the values in question are "big" (e.g. long strings or complex class objects), that could be time consuming. But arguments which are passed to reference parameters don't have their values copied.