Dangling Pointers

coursera.org/learn/pointers-arrays-recursion/supplement/X78Fx/dangling-pointers

When you write code with arrays, you may be tempted to return an array from a function (after all, it is natural to solve problems where an array is your answer). However, we must be careful, because the storage space for the arrays we have created in this chapter live in the stack frame, and thus are deallocated after the function returns. The value of the expression that names the array is just a pointer to that space, so all that gets copied to the calling function is an arrow pointing at something that no longer exists. Whenever you have a pointer to something whose memory has been deallocated, it is called a *dangling pointer*. Dereferencing a dangling pointer results in undefined behavior (and thus represents a serious problem with your code) because you have no idea what values are at the end of the arrow.

If we were to try to compile the code in the video, the compiler would warn us that our code is broken if we used this problematic behavior:

```
1

2

warning: function returns address of local variable [-Wreturn-local-addr]

   return myArray;
```

However, you should understand why the compiler is giving this warning, and never write code which exhibits this bad behavior, rather than relying on the compiler to find it. The compiler's ability to detect this sort of problem is rather limited. Consider the following code, which is still broken, and has the exact same effect as the code in the video:

```
1

2

3

4

5

6
```

```
7

8

9

// Still broken

int * initArray(int howLarge) {

  int myArray[howLarge];

  for (int i = 0; i < howLarge; i++) {

    myArray[i] = i;

  }

  int * p = myArray;

  return p;

}
```

gcc produces no warning, even though the code is equally broken. If you execute a call to this function by hand (recommended), you will find that its return value is dangling.

One thing to be particularly wary of with respect to dangling pointers is that *sometimes* you can get away with using them without observing any problematic effects—your code is still broken, it just does not look (or even behave) broken. Having the code seem fine is particularly dangerous for two reasons. First, when there is a problem in our code, we want to find it. We want to fix it, so that our code is correct, and we do not have danger lurking inside. Second, the "but I did that before and it was fine" effect is dangerous to novices—if you learn that something is ok, you keep doing it. You do not want to form bad habits.

To understand why you only see problems sometimes, remember that a value stored in memory will remain there until changed by the program—and a deallocated memory location may not be reused immediately. Once a function returns and its frame is destroyed, the memory locations that were part of that frame will not be reused until more values must be placed on the stack. If we call another function, its frame will be placed in the next available stack slots, overwriting the recently deallocated memory. Only at this point will the values associated with the deallocated stack frame change (due to assignments to variables in the new frame). Now writing to memory through the dangling pointer will change variables in that frame in unpredictable ways. Note that is *not* safe to

use a dangling pointer into a deallocated frame even when you have not called another function—even though most stack allocations will come from calling a function, there is nothing to guarantee that those are the only way that the stack is used.

Completed