C: A Language Built Around a Memory Model

Unlike Java or Python, C is a language built around the idea of *manual* management of computer resources. This means that handling the lifetime of a resource is the programmer's responsibility. In C, the most prominent of those resources is memory.

Storage Duration

When declaring variables in C, you need to explicitly think about the *duration* of your data: is it short-lived or long-lived?

Local (aka *automatic*) storage duration is the default, and local memory used to store data is automatically reclaimed (``deallocated'') whenever the enclosing scope is popped off the runtime stack. Local data is therefore "short-lived."

Allocated data must be explicitly requested and is only deallocated when deallocation is requested explicitly by the programmer. Allocated data is therefore "long-lived," since it persists until it is either manually deallocated by the programmer or the program terminates.

Requesting local storage

Local memory is automatically allocated whenever a variable is declared. For example,

```
int x;
```

reserves space for an integer.

```
int x = 23;
```

actually does two things: 1. it reserves memory (usually on the runtime stack), and 2. it stores the value 23 in that memory.

If our program had the following code:

```
void foo() {
    int x = 23;
}
```

then `x` would be automatically deallocated at the end of `foo`, when `foo` returns control to the calling function (whatever that is).

Although C is allowed to store local data in a variety of places, *it is almost always stored directly on the runtime call stack*. C programmers sometimes say that a variable is "on the stack." What they really mean is that the variable is "local," and you will probably catch me saying this every now and then. We also sometimes just call them "locals."

Requesting allocated storage

Allocated memory is manually managed. For example,

```
int *x = malloc(sizeof(int));
```

allocates space for an integer. `malloc` is a standard library function, so you must `#include <stdlib.h>` in order to use it. `malloc` takes the size of the data type, in bytes, as its sole argument, and it returns a pointer (i.e., an address) to that memory.

Although C is allowed to store allocated data in a variety of places, *it is almost always stored in the heap*. What is "the heap"? Think of it as whatever memory is not being actively used by the program to manage itself. For example, the call stack is used to manage the execution of functions, so the stack is not the heap. C programmers sometimes say that a variable with allocated duration is "on the heap," and you will probably hear me say this as well. What they really mean is that a local variable stores a pointer to heap storage.

Look at the last code example again. There are actually *two* allocations happening. Can you spot them? It's easier to see if we split the allocation and the assignment into two pieces, ala

```
int *x;
x = malloc(sizeof(int));
```

Here, we first allocate a local variable `x` (on the stack). `x` stores a value of type "pointer to int". Then we ask the operating system, via the standard library function call `malloc`, to give us enough memory (on the heap) to store an `int`. Finally, we assign the pointer our requested (heap) memory to `x`.

Wait... pointers?

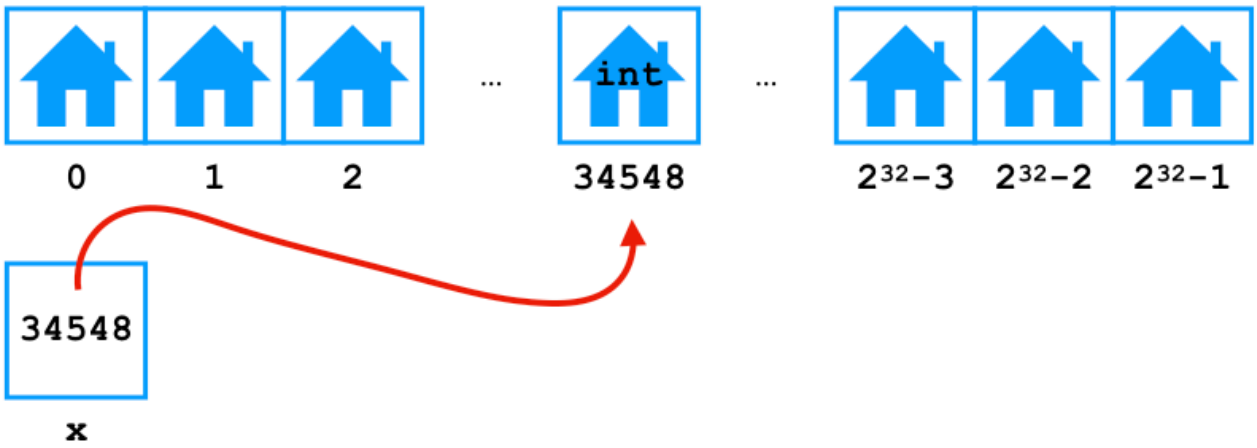
Despite the hype, pointers are actually very simple. It's their simplicity that usually trips people up, because you can use their simple features in complex ways that can get confusing. But really, keep in mind that they are simple and follow simple rules.

A *pointer* is just a memory address. That's almost the entire story.

When working with pointers, you usually want to do one of two things:

1. *Follow* a pointer to the data it points to, or
2. *Get a pointer* to a value.

The first operation, following a pointer to the data it points to, is called a *dereference*. This sounds a little frightening, but really, if you imagine a pointer as being like an address to someone's house, written on a slip of paper, dereferencing is just walking down the street to the address where the house is located. Fortunately for us, in memory-land, all values live on one street, with address `0` at the beginning of the street and address $2^{32} - 1$ at the end.



In our `malloc` example above, you'd find an `int` living at the address written on the piece of paper `x`. And because `x` got the address for `int` from `malloc`, we know that the address to `int` is probably somewhere in the heap.

For example, let's dereference `x` and store a value there.

```
int *x = malloc(sizeof(int));
*x = 3;
printf("%d", *x);
```

The above program will print 3.

The second operation, getting a pointer, is called *address of*. It does exactly what it says it does: it gets the address of the thing you're asking about. For example,

```
int *x = malloc(sizeof(int));
*x = 3;
int *y = &(*x);
printf("%d", *y);
```

What do you think this program will print? It prints 3.

1. On line 1, we allocate memory for an `int` on the heap and store a pointer to that memory in `x`.
2. On line 2, we follow `x` to its location (i.e., we dereference `x`) and then we store 3 *in that location*.
3. On line 3, we dereference `x`, obtaining a value stored in the heap, but then we immediately ask for the value's address using `&`. We then store this address in `y`, which is a pointer.
4. On line 4, we print the value pointed to by `y`.

If you are not convinced that `x == y`, try this:

```
int *x = malloc(sizeof(int));
*x = 3;
int *y = &(*x);
printf("%p == %p ? %s\n", x, y, x == y ? "yes" : "no");
```

On my machine, when I run this program, I get output like:

```
0x7ff124400350 == 0x7ff124400350 ? yes
```

The confusing part about pointers is that we use `*` in two contexts:

1. In the type declaration of a variable, e.g.

   ```
   int *ptr;
   ```

2. And when dereferencing a variable, e.g.

   ```
   int foo = *ptr;
   ```

So you need to pay attention to which context you're in, otherwise you'll get it wrong.

`malloc` may fail

One important thing to note is that calls to `malloc` can fail. Why? There are many reasons that this may occur, but all of them fundamentally boil down to the fact that sometimes the operating system cannot find enough memory to satisfy your request. When the failure occurs, `malloc` returns `NULL`. You should get into the habit of checking that `malloc` does not return `NULL`.

```
int *x;
x = malloc(sizeof(int));
if (x == NULL) {
    // do some recovery action; sometimes
    // the best thing to do is to kill the program,
    // returning a "failure" code to the OS.
    exit(1);
};
```

Assuming that your allocation was successful, in order to assign a value to that memory, we need to *dereference the pointer*. We dereference using the `*` operator. For example, the following dereferences `x` and then assigns `23` to the location pointed to by `x`.

```
*x = 23;
```

Returning to our `foo` function with some small modifications,

```
void foo() {
    int *x = malloc(sizeof(int));
    if (x == NULL) {
     exit(1);
    }
    *x = 23;
}
```

We now have a value (`23`) in memory (at address `x`) that behaves very differently than the local version: when `foo` ends, and the function returns control to its caller, the memory pointed to by `x` remains allocated.

Why? Because it has "allocated duration" and you did not tell C that you no longer needed that memory. In fact, we have a little problem with this particular program: after `foo` returns, not only can we not access the value `23` (`x`, the pointer, is local to `foo`), the pointer value is *effectively gone* when the function returns. We've lost the address. Without the address, we can't tell C to *deallocate* the `int` stored at `x`!

This kind of programming mistake has a name in C: it's called a *memory leak*. Memory leaks are an easy mistake to make in C. If you leak

enough memory, eventually your program runs out of it, `malloc` will eventually return `NULL`, and at that point, your program is toast.

Fortunately, the fix for this program is simple: Use `free`.

Like `malloc`, `free` is also a standard library function.

```
void foo() {
    int *x = malloc(sizeof(int));
    if (x == NULL) {
     exit(1);
    }
    *x = 23;
    free(x);
}
```

Now `foo` doesn't suffer from the memory leak. Of course, `foo` has other problems, like... it doesn't actually do anything... but that's OK for now `;)`

It's not always easy to know when to `free` memory, and so most memory leaks are not simple ones like the one I showed you above. Still, if you keep in mind the rule that "every `malloc` should be accompanied by a `free`", you'll be off to a good start.

When should I use allocated storage?

You might be thinking: "All this manual memory management sounds like a lot of work! Do I really need to use it?" Trust me, I thought exactly the same thing the first time I heard about this, too. The short answer is yes, you have to use it.

One of the big advantages of languages like Java or Python over C is the fact that all memory management is automatic. In fact, automatic memory management techniques were already known when C was invented. So why did C's inventors make it manual? There are at least two reasons:

1. Manual memory management is a *feature* in C. Remember that C was written with UNIX in mind: the designers of UNIX needed direct access to memory because an operating system needs to be able to speak directly to hardware. Code that manages the interaction between hardware and an operating system is called a *device driver*. Knowing exactly when to *automatically reclaim* memory is tricky in the context of device drivers.

2. Manual memory management *can be* more efficient than automatic memory management. It should be noted, though, that while this is

indeed a true statement, the performance penalty for using automatic memory management in a modern language on a modern computer is often negligible, and not worth the pain of manual management. When C was invented in the early 1970's, with much slower computers, manual memory management was a better value.

Getting back to the question, "when should I use it?", the short answer is: whenever a value needs to outlive the scope in which it was created. That sounds a little cryptic, so here's a concrete example:

```
??? zero_fill(int length) {
    // create an array of length n, filled with zeros
    ...
    ...
    ...
    return ???;
}
```

I want a function that allocates an array of length `n`, fills the array with `0`s, and returns it. Notice that I left the return type and value unspecified (`???`). So what's wrong with this version?

```
int[] zero_fill(int length) {
    int arr[length];
    for (int i = 0; i < length; i++) {
      arr[i] = 0;
    }
    return arr;
}
```

Well, aside from the fact that it does not compile (`int[]` is not a valid return type), the problem is that we just allocated `arr` in memory local to `zero_fill`.

More importantly, how *would* this work? Assuming that the compiler accepted the above, how might we imagine this working?

Let's say that `main` calls `zero_fill` so that `zero_fill` is the subroutine at the top of the stack (Fig. 4).

Since `arr` is declared as an automatic variable, the entire array is allocated on the stack, in the stack frame for `zero_fill` (Fig. 5). This is part of what we mean when we say that `arr` is *local* to `zero_fill`.

`zero_fill` is supposed to return `arr` to `main`. *How* might we return it?

Let's say that we return a copy of `arr`. Now we have a problem: only `zero_fill` knows how big that array is going to be. Do we have enough space in `main` to store the copy? Probably not! (Fig 6)

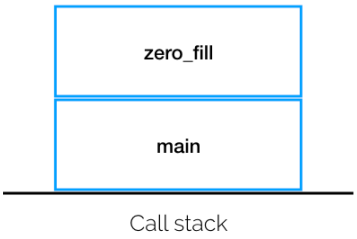


Figure 4: A call stack with `zero_fill` as the active subroutine.

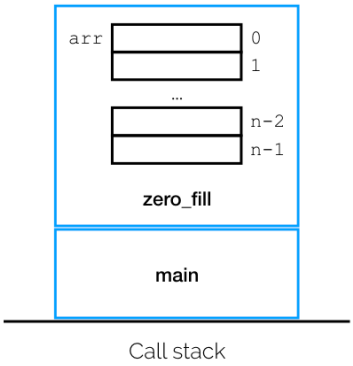


Figure 5: `arr` is allocated inside `zero_fill`'s stack frame.

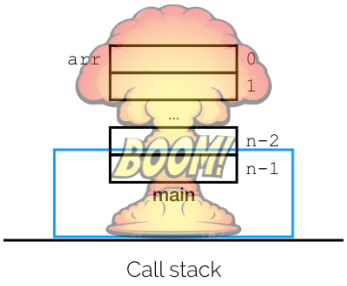


Figure 6: If `zero_fill` returns a *copy* of `arr` to `main`, it might not fit in `main`'s stack frame.

Even if we insist that we want it to work this way, is this really what we want? We already did the work of creating `arr`. Are we really going to *make a copy* of it? What if `arr` has a million elements? Copying it might take a long time.

The alternative approach is that we *don't* copy `arr`. Instead, we return the *address* of `arr`. In other words, we return a pointer to `arr`.

```
int* zero_fill(int length) {
    int arr[length];
    for (int i = 0; i < length; i++) {
      arr[i] = 0;
    }
    return arr;
}
```

This is a much better arrangement, and indeed, this program even compiles. It solves two of our problems: a pointer is small (e.g., 4 bytes) and we know exactly how big it will be ahead of time, so we can copy it back into `main` quickly.

But there is another nasty problem. What does `ptr_to_arr` in `main` point to? It points to memory *local* to `zero_fill`.

When `zero_fill` returns, C reclaims `zero_fill`'s memory by popping it off the stack. If we dereference `ptr_to_arr` after `zero_fill` returns, just about anything could happen because that memory is free for the application to use for other purposes (Fig. 7).

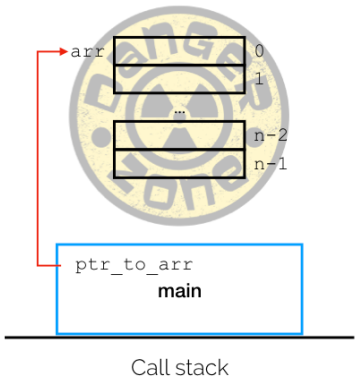


Figure 7: Pointers to deallocated memory are a bad idea.

C, by the way, will happily let you write this program. A good compiler (like `clang`) will warn you, but it is perfectly valid C. Worse, it might even work for you when you test it. But this problem is serious enough that it has a name: it is called a *dangling pointer*. In this case, dereferencing this dangling pointer will use memory that has been "freed" by the call stack; therefore, this bug is called a *use-after-free* bug.

To get around this bug, we need to use memory with *allocated duration*. In other words, we need to use the heap. Here is a correct program:

```
int* zero_fill(int length) {
    int *arr = malloc(length * sizeof(int));
    for (int i = 0; i < length; i++) {
      arr[i] = 0;
    }
    return arr;
}
```

Note that I've omitted the `NULL` checks after malloc for clarity, but for completeness, you really should check.

Now we've allocated an array on the heap. `arr` is still a local variable, but it points to memory on the heap:

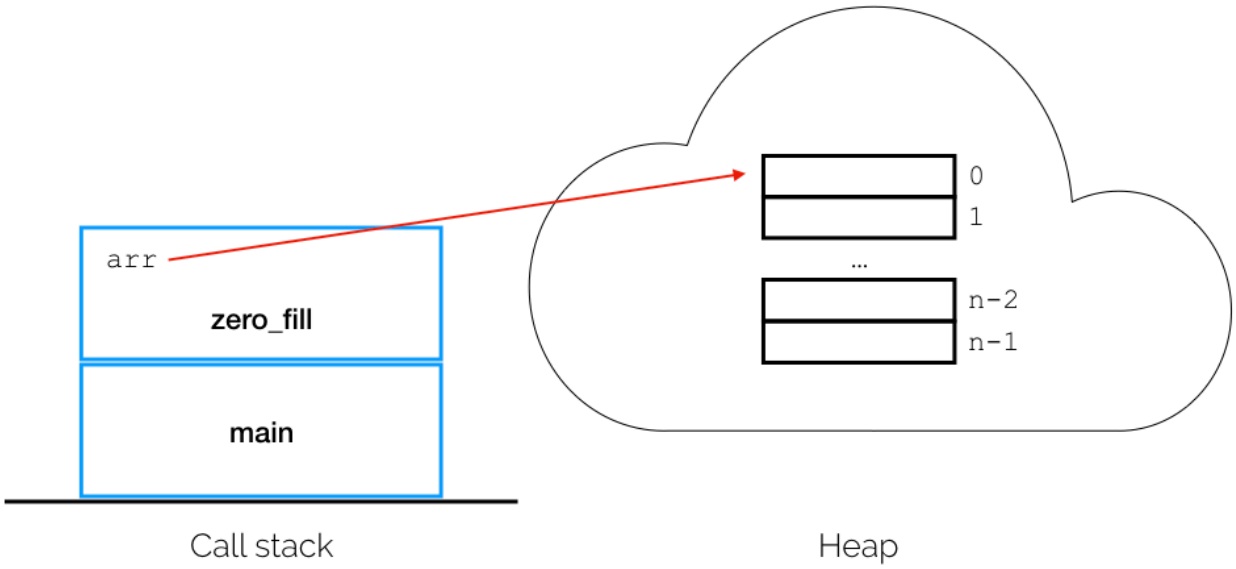


Figure 8: Totally cool use of memory.

When we return `arr`, C *copies* the value of `arr` (an address) into whatever local variable we've decided to put the return value in `main` (e.g., `ptr_to_arr`). When `zero_fill` is popped off the call stack, deallocating the local `arr`, our program is unaffected by the deallocation (Fig. 9).

The gotcha with allocated duration storage is that we need to remember to `free` `ptr_to_arr`, otherwise we leak memory and may eventually run out of memory.

```
int main() {
    int *ptr_to_arr = zero_fill(2000);
    // ... do other things ...
    free(ptr_to_arr); // we remember to free!
}
```

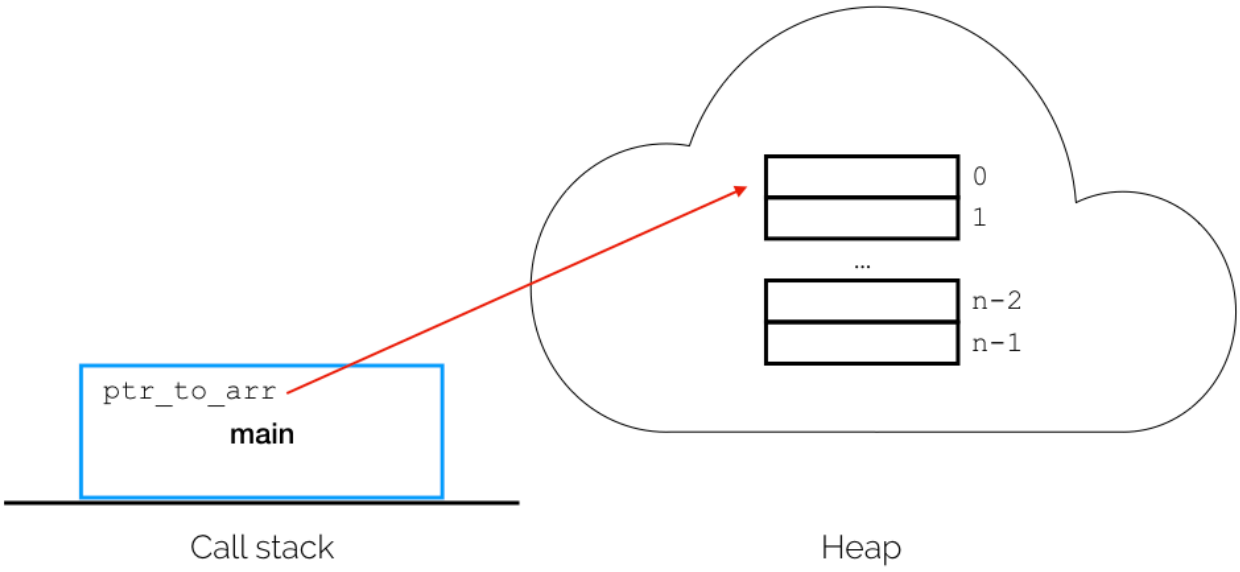


Figure 9: Everything is still totally cool.