Memory Management in C

C is a “close to the metal” language. This means that you can access the hardware in a very “native” way. One of the biggest ways this manifests itself is through the pointer-integer duality. In understanding the pointer integer duality, you will start to understand the foundation of memory management in C. Furthermore, understanding memory management in C will go a long way towards helping you understand the memory model of other “higher-level” programming languages.

[Disclaimer] This is meant to be for relative beginners in C, so take everything with a grain of salt.

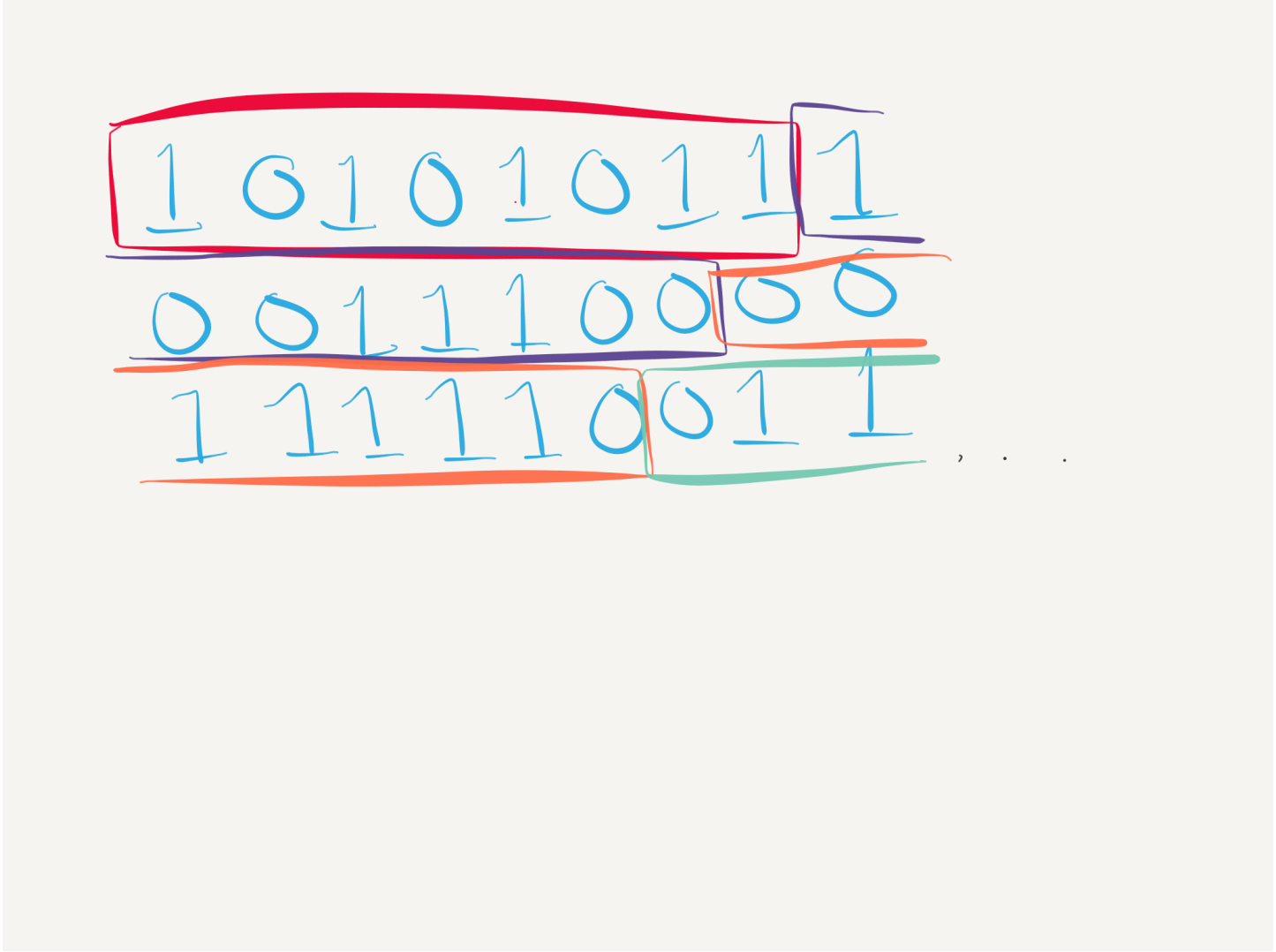
Pointer-Integer Duality

Understanding the Address Space

Binary Data to Bytes

Computer memory at its basis can be thought of as a lot of 1s and 0s. This binary data is formless and meaningless (and thus useless) unless we add some structure on top of it. So what we can do is combine 8 of these 1s and 0s together to form bytes. Then we can think of each byte as representing an integer between 0–255 and can think of all these 1s and 0s as a giant array of bytes called `Memory`. Notice, however, that these decisions that we made were relatively arbitrary, this idea will come back when we look at memory in C specifically. C takes advantage of the idea that data can be interpreted in many different ways.

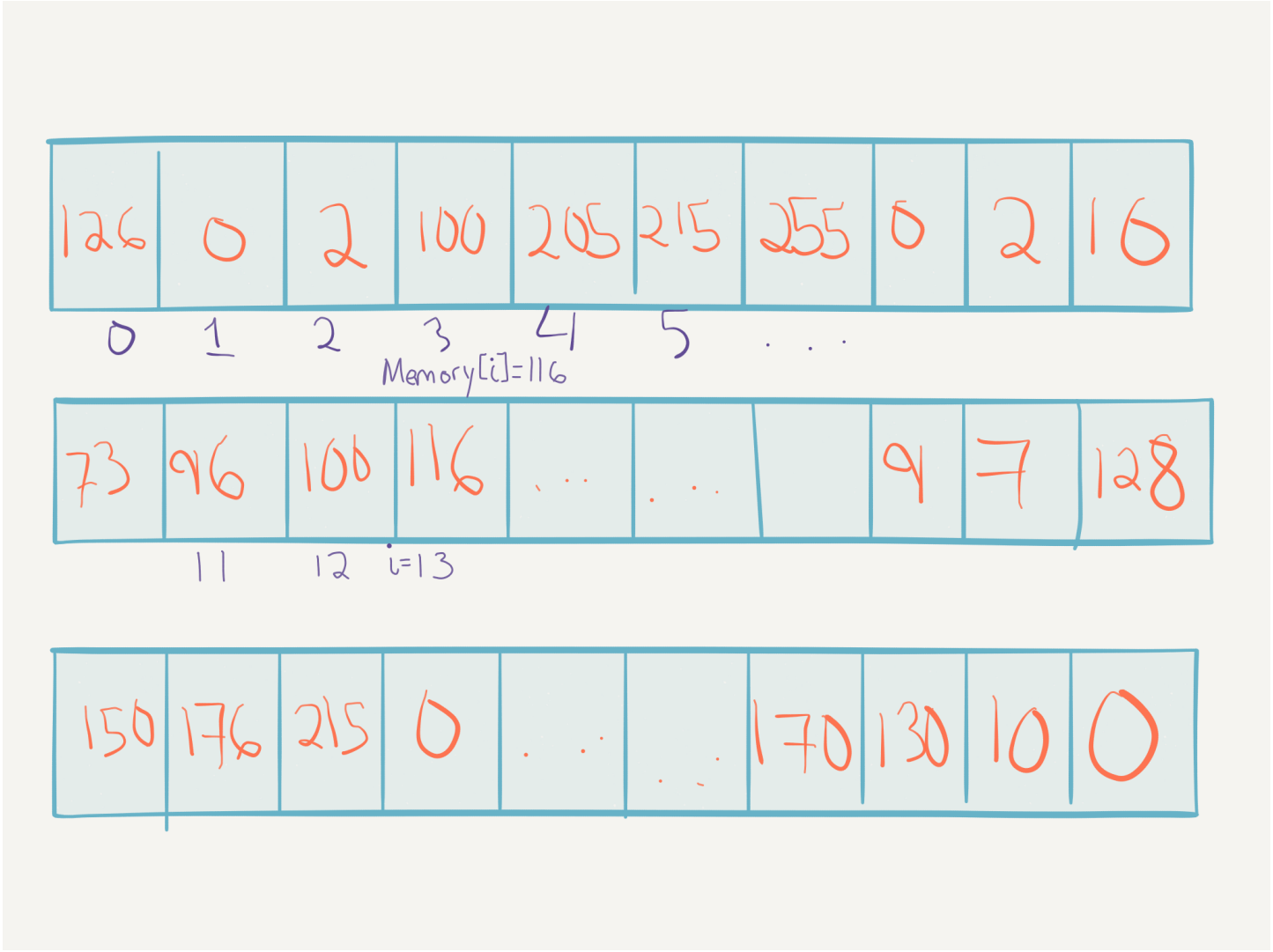
In this diagram the boxes indicate bytes, and the 1s and 0s in these boxes are bits:



One Giant Array

So now that we can think of the entire address space as a giant array of bytes, we can now start to reason about it. For

instance, how do we select a single byte in this `Memory` array? If we have a computer that has 16 Gb of RAM then there are 16,000,000,000 possible elements. This means that our index `i`is logically bounded between: `0 <= i < 16,000,000,000` and can be used to select one byte (`b`)from this array: `Memory[i]`. Given `i` we can recover the byte at `Memory[i]`, therefore this index `i` "points to" byte `b`.



Index-Pointer Transformation

Although this index `i`does not seem to be a pointer if we add a little syntactic sugar this relationship becomes clear. The giant array of bytes is called `Memory` and there is an index into this array called `p`. Normally we are used to seeing `A[i]` to give us the ith element in an array `A`. However, pointers are used so much in C that it would be a pain to expose a global `Memory` array and have it always accessed by saying `Memory[i]`.To help us with this we add some syntactic sugar. We create a new type called a `pointer`, a `T*` where `T` is a type in C and the `*` indicates that it is a pointer to an element of type `T`. It is defined such that, for a pointer `p`, writing`*p` is the same thing as `Memory[p]`. A pointer is just an index into this global memory array! This has the side effect: A pointer is just an integer.

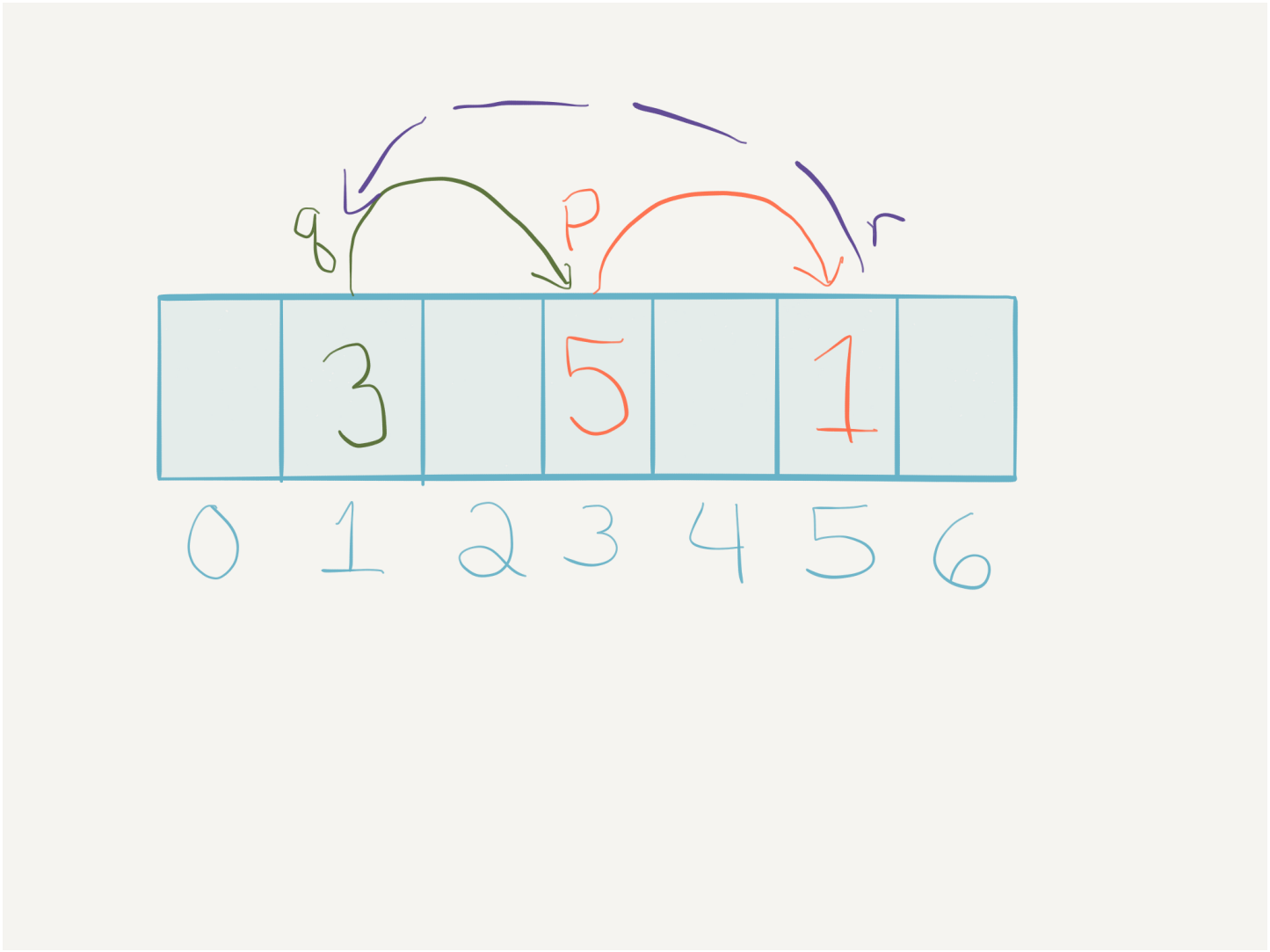
Values to Pointers

Whenever you have a value `v` in C it has to exist somewhere in this `Memory`array that we mentioned earlier. In order to find out where it is in memory all we have to do is write `&v`.This gives us the "index" `i` where the element `v` is in the array `Memory`. This means `Memory[&v] == v` and thus `*&v == v`. `&` and `*` are inverse operators in this sense, where `&` takes the "address" of a value and then `*` finds out what is at a given address.

Meta Pointers

Since pointers are just integers (and therefore values) we can take a pointer to a pointer to a value, and so on. We can have a pointer `p = 5` such that `*p = 1` and `p` is stored at index `3` in `Memory`. And we can have another pointer `q = 3` such that `*q = 5` where `q` is stored at index `1`. This means that `p` points to the number `1`, and `q` points to `p`(which is stored in memory location `5`). In this sense a pointer is just a number. The number `1`that `*p`is equal to, could just be anything, a number to indicate the number of variables in an array or a the number of children a node in a tree has, or it could be another pointer

(`r`) back to `q`(since `q`is stored in memory location `1`. `Memory[i]` for any index `i` could be a value (like an integer) or it could be a pointer (which is a different type of value).



Please make sure you understand this example before moving forward.

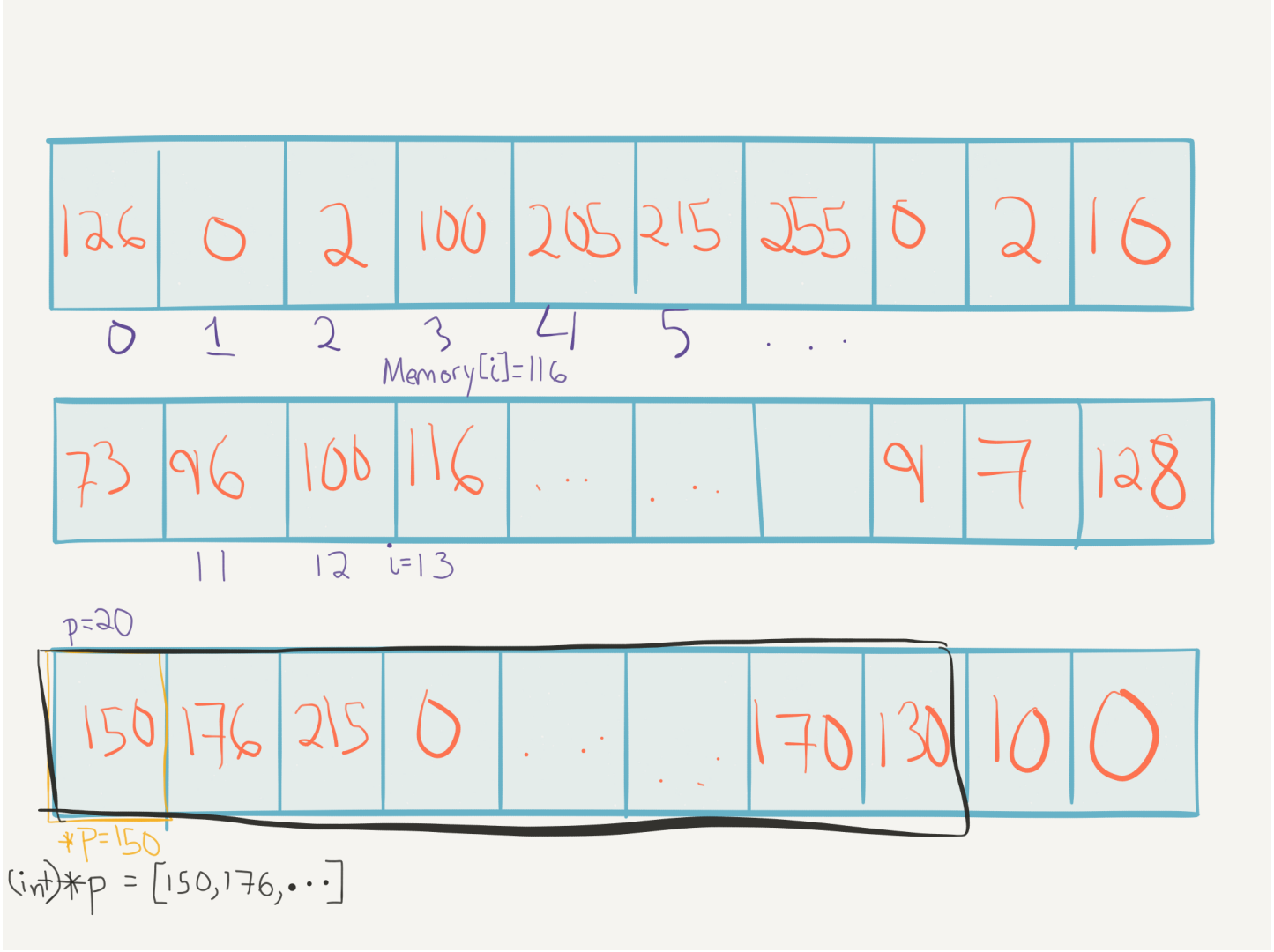
Type Casting

So now we go back to our initial assumptions where `Memory`was a giant array of bytes. However, if this is true `Memory[i]`would always point to a byte! But sometimes we want to point to integers, pointers, structures, etc. Instead we can look at `Memory+i` where `Memory+i == &Memory[i]`. `Memory+i` is a pointer to the ith byte in our memory array. This means that `*(Memory+i)` is the same thing as saying `Memory[i]`.

We will name a pointer `p` such that `p == &Memory[i]`. Now that we have this pointer `p` let's go back to our initial assumption. The only reason why `Memory[i]` gave us a byte is because we told it to only return `8` bits or `1` byte implicitly. This is because we have implicitly defined `Memory` to be of type `char *`. However, this "type" tells us that `Memory` is an array and that each element is of type `char`, and therefore we should only read one byte when looking at `Memory[i]`.However we can force C to give us a different element than a byte by writing: `(int) *p`.This tells C to treat what p is pointing to as an integer...

Now it doesn't actually matter whether or not `*p` is an integer or not, ultimately it just a location in memory with a bunch of 1s and 0s after it. We are telling C to think of the bits that follow as an integer. For C this means `(int) *p`is 64 bits (8 bytes, the size of an integer) and these 64 bits are treated of like bits in an integer. This is called a "type cast" and can be used to say that any data should be interpreted as this type.

When we declare a pointer `int *p`we say this pointer `p`points to an integer. This means that when we do `*p`to dereference this pointer, C will give us an integer rather than a byte. That is because C essentially says that `*p`instead of "desugaring" to `(char) Memory[p]`, it instead, desugars to `(int) Memory[p]`.C does all of this "type casting" for us.



Pointer Arithmetic

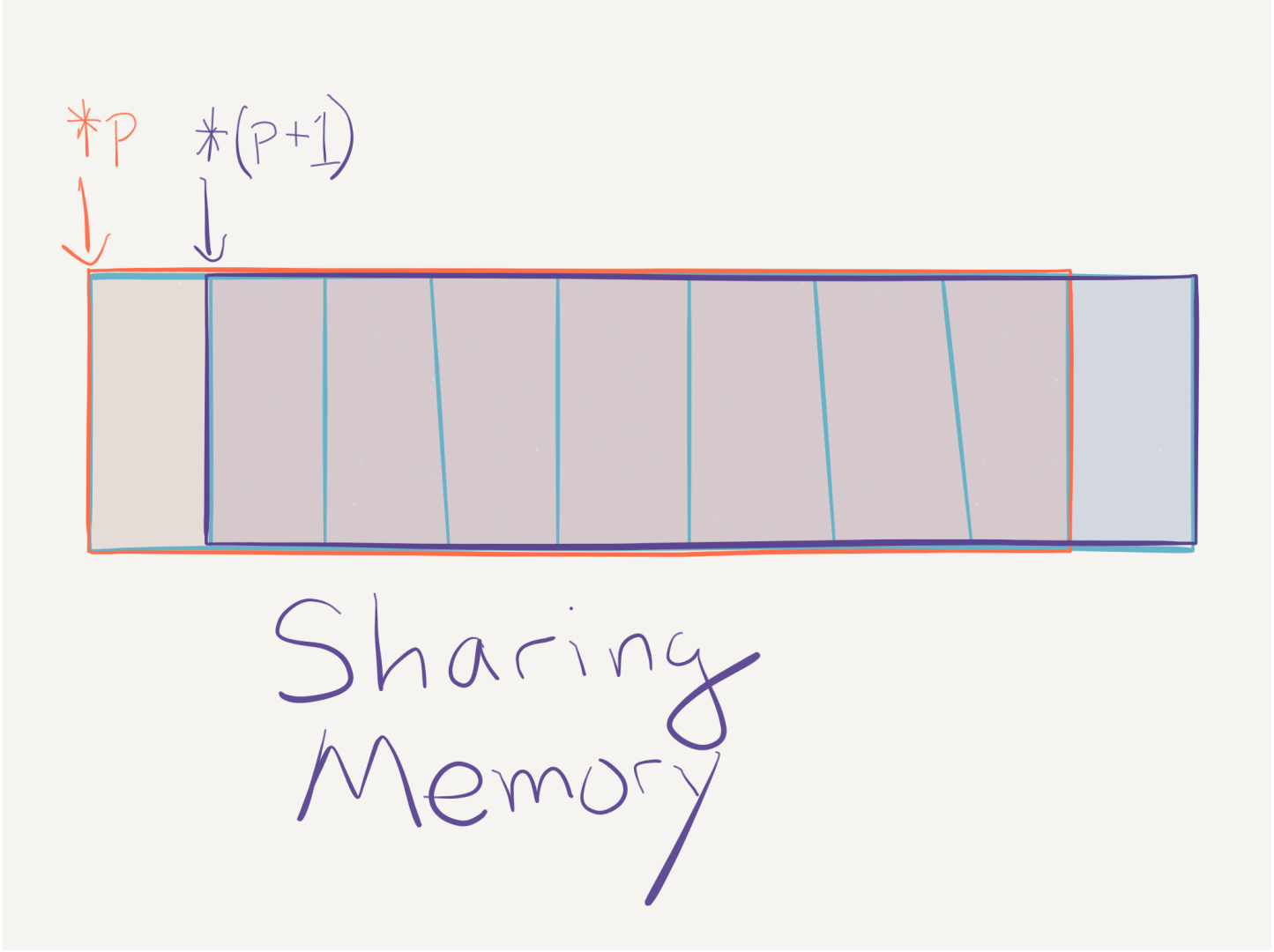
We know that we can declare multiple different types of pointers `int *p, char *q, struct Vertex *s, etc.` We also know that pointers in C are just integers. Normally when you add an integer `i` to `j`, you get `i + j`. However, just as we added special rules to pointers for when you say `*p` we also add special rules to pointers when you perform arithmetic (+/-) on them.

This is similar to how you when you add `x kilometers` to `y meters`, you have to normalize both `x` and `y` to be in the same unit. In C, types are units.

If pointer arithmetic was the exact same as unitless integer arithmetic it would be bad if you wrote: `long int *p; *p = 4; *(p+1) = 3`. You might think that this would set two adjacent integers to 4 and 3 respectively, however, these two pointer `*p` and `*(p+1)` share memory. Since `p` is an long integer we need 8 bytes starting at `Memory[p]`. These next 8 bytes are used to represent the given number `4`. However `Memory[p+1]`, would start just one byte after `Memory[p]`and then share 7 bytes with `*p`! This means that `*(p+1) = 3` would reset some of the same bytes that `*p = 4` set. Instead you would have to write `long int *p; *p = 4; *(p+8) = 3`. And change what you increment `p` by on a per type basis.

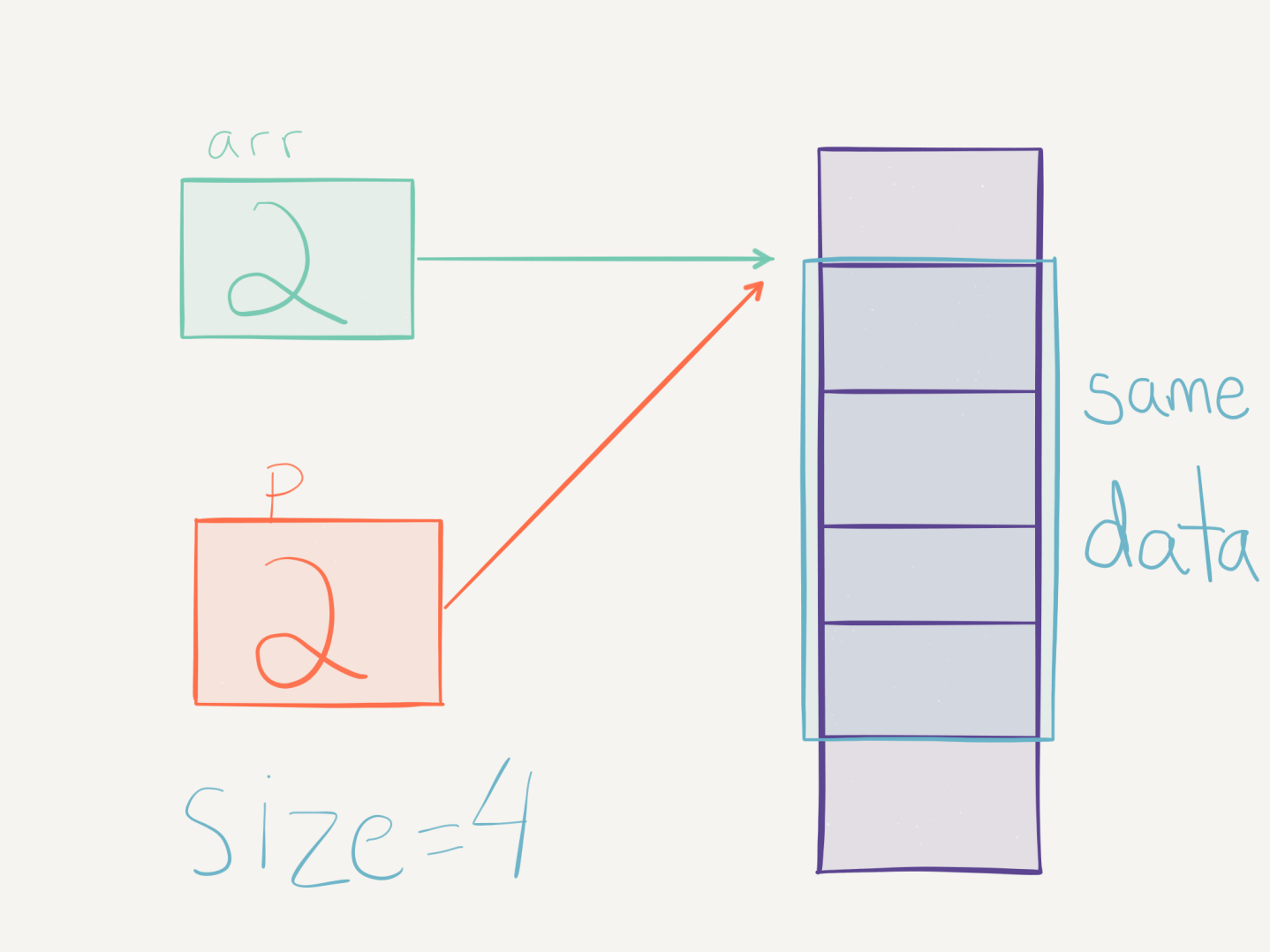
Instead pointer arithmetic is like arithmetic with types as units. So when you do `p+1`, C asks what the size of the type underlying p (`long int`) is (`8`), then adds that to `p`: essentially becoming `p + sizeof(*p)` (under unitless arithmetic rules). Just like if you add `1 kilometer` to `10 meters`, you first say "how long is a kilometer in meters?", then convert kilometers to meters, then add the two, getting `1010 meters`. Therefore,`p+1`, where `p` is a pointer to an `long int`, points 8 bytes after `p`, and if `p` is a pointer to an `int`, 4 bytes after `p`. `p+4`, wherep is a `long int`, is 4*8 bytes ahead of `p` (32 bytes).

If we were to use unitless (regular) arithmetic instead of pointer arithmetic for the equation `long int *p; *p = 4; *(p+1) = 3`:



Pointer-Array Duality

Now that we know about “type casting” and how pointer arithmetic works, we can look at arrays. Arrays are meant to specify a contiguous sequence of data. We know that we can access an element of data by saying `*p` and we can point to this element by saying `p`.Therefore all we need is a convention for arrays. In our case we just say `p`points to the beginning of an array and `p[i] == p+i` with the pointer arithmetic rules we talked about in the last section. There! We have array notation, all they are are pointers, which are just indices into the global memory array, which are just integers! But notice that an array `arr = p`,does not have any notion of size directly attached to it, to do that you will have to keep track of the size separately (in an integer, or through putting a sign at the end, etc.). Furthermore `arr = p` does not copy the data that `p` points to, it just copies the integer `p`, and therefore it points to the same memory. This means that if you change `arr[i]`, then `p[i]` will change and vice-versa. Then you can access any `A[i]`such that `0 <= i < size`, where `size` is the size of the array `A`.



The Null Pointer Convention

So now we know how to "address" values (we know where we can find them, hence the 'address'). But what happens if we don't actually want to point to anything. Since pointers are just integers with these special syntactic sugars mentioned in the previous sections, we can't just let this this "integer" just be a random number. If this was the case then this "integer" when thought of as a pointer (or an index into the global memory array), would point us to a random memory location, rather than telling us that it is invalid. So for a pointer to be invalid in C, we use the null pointer convention. In C a null pointer is typed as `NULL` and is just the integer `0`. This means that whenever a pointer points to the memory location `0` in `Memory` we just say that it is invalid. Of course that means that we can't use the value in `Memory[0]`. But that just means we "only" have `16,000,000,000 - 1`bytes that we can use... so sad.

Therefore in C whenever there is a pointer `p == 0`and you try to dereference it (`*p`) C will throw a null pointer exception saying you tried to dereference a null pointer. Since we have all agreed that there is nothing useful in `Memory[0]` accessing it is invalid.

Strings

As we mentioned in the Pointer-Array Duality section, every array can be thought of as a pointer. However, we need a way of knowing how long this array is, how much memory is in this array. We mentioned earlier that you can do this by storing the size of the array in a separate integer. However in earlier days, memory on computers was scarce, so using a whole integer for determining the length of the array was too much, especially for something as common as strings. To get around this, they adopted the convention in strings (much like they did in pointers), that the `0` byte (the null byte/terminator, `\0`) is actually a sign. In strings in C the "null terminator" is used to indicate where the end of the string is. As long as the string contains no other `\0` bytes you can tell how long the string is by reading it until you reach a `\0`.

This is why, if a non-null terminated string (a string without `\0` at the end) is passed to a function expecting a null terminated string like `printf("%s", string), strlen, strcmp, ...`,then we will read past the end of the string into invalid memory. Our array has no information regarding its size! It is now effectively an infinite length array starting where `string` points, and these functions don't know where it should end. The null terminator is a source of endless bugs. Today, when developing

your own code, using an explicit size (stored in an integer) could make things easier, but when passing into C functions you didn't write (like `strlen`) always make sure your string is null terminated.

Memory Management Functions in C

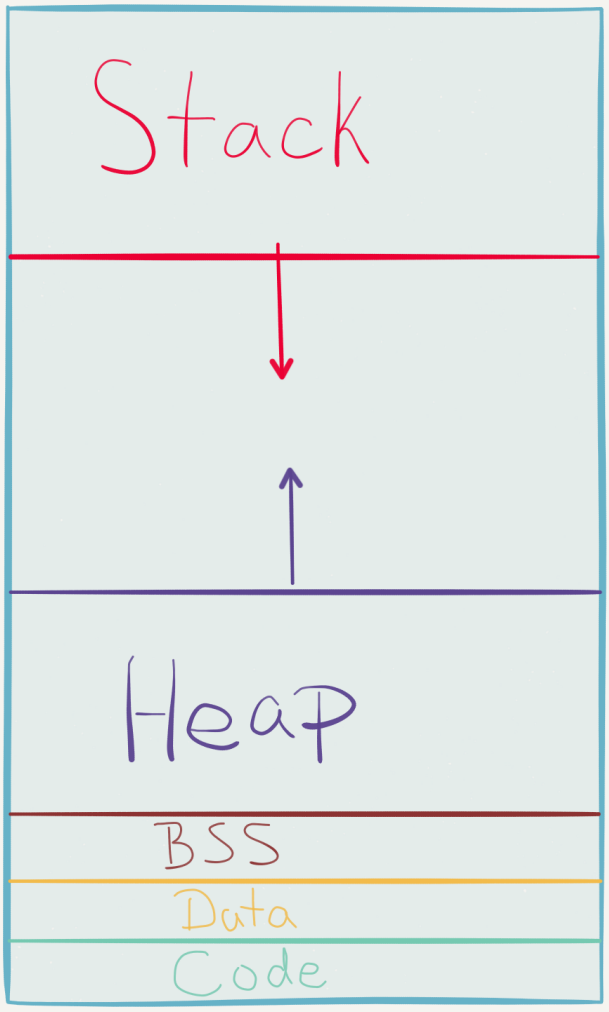
It would be unreasonable for most people to deal with this universal memory array. What bytes are valid? What is being used? How can we enforce that accessing "invalid" memory causes the program to crash, rather than mysteriously continue. To deal with these potential complexities, we use the abstractions of the Stack and the Heap.

Overview:

The stack is where function data is stored. This includes, local variables, variables to connect one function to another, etc. The Stack "grows down" in memory towards the Heap.

The Heap on the other hand stores variables that we have "malloc'd" or "calloc'd". This heap memory grows up.

The other sections of memory store statically initialized data like string constants and your code.



Stack

Functions and Local Variables

In C we have functions. These functions are located in memory on something call the `Stack`, not to be confused with a `stack` (data-structure). The `Stack`is a `stack`where functions are stored. Each element on this stack is a "stack frame" which is responsible for maintaining all of the local variables in a given function as well as the information to return to the function that called it.

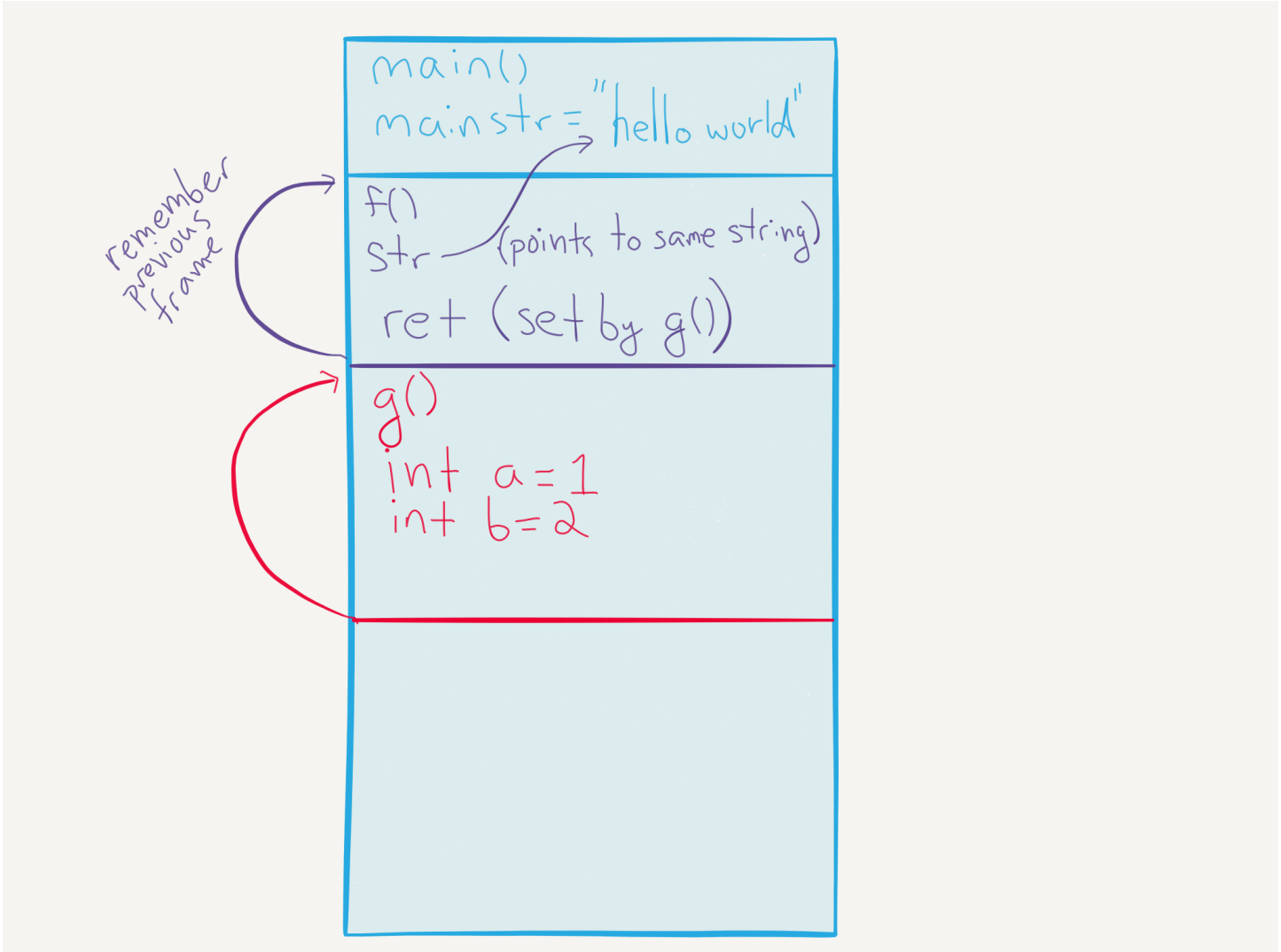
In order to visualize this stack let's imagine there are three functions `void main()` `void f(char * str)`and `int g(int`

`arg1, int arg2)`. `main` has the local variable `char* mainstr="hello world"`, `g` has the local variable `int a` and `g` has the local variable `int b`.The stack is organized into stack frames. Each frame is responsible for containing all of the information needed in a function. In this case if `main()` calls `f()` and `f()`calls `g()`, there are three stack frames, `MAIN`, `F` and `G`. The stack frame main contains space in it for one local variable `mainstr`, F contains space in it for one local variable `a` and a pointer to `MAIN`, and `G` contains space for one local variable `int a` and a pointer to `F`.

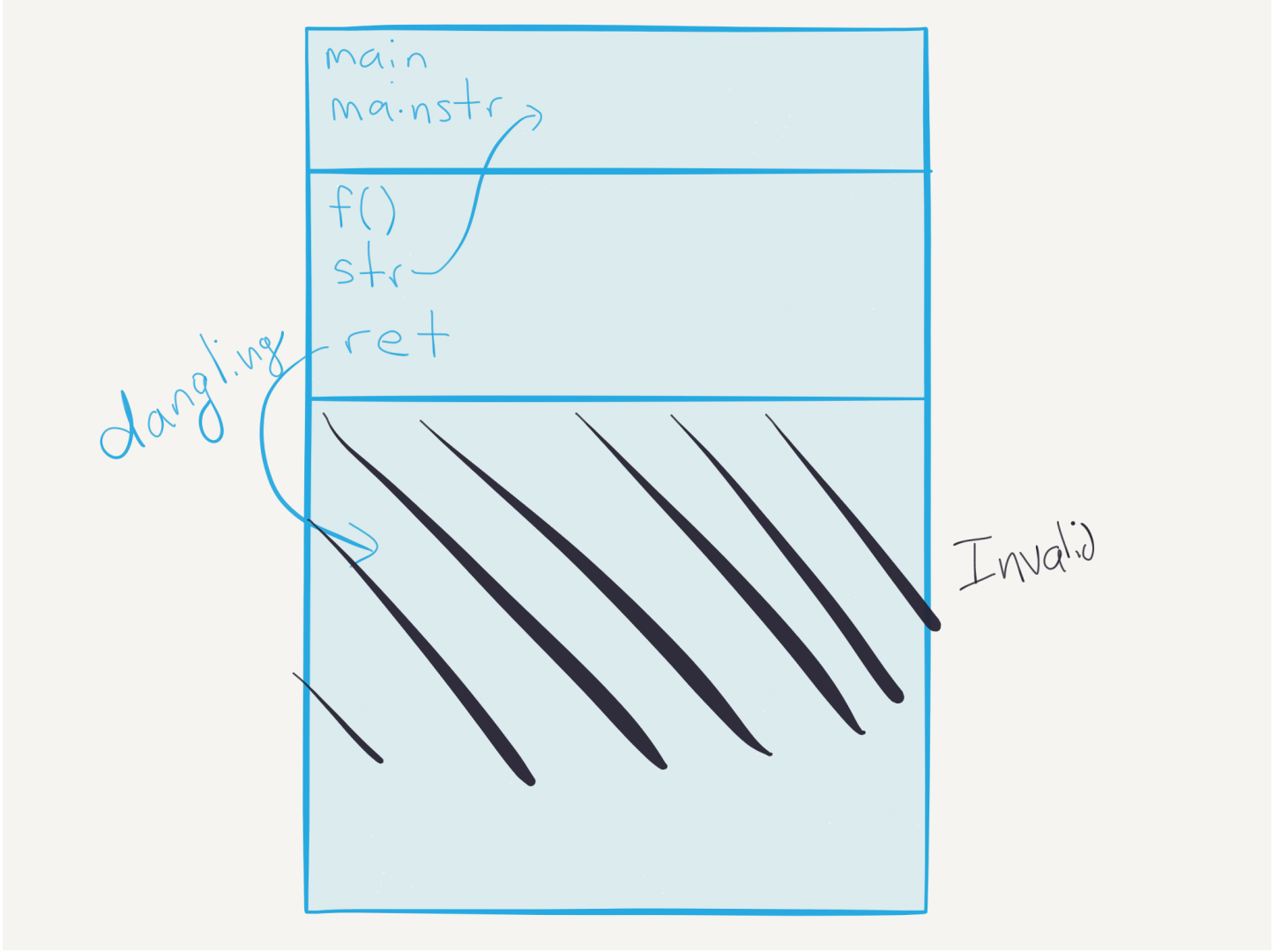
Initially there is only the `MAIN` stack frame. Then when `main()` calls `f(mainstr)` the stackframe `F` is pushed onto the call stack with the argument `str=mainstr` and space for the return value of `g()`. After this `f()` calls`g(1, 2)` and the `G` stack frame is pushed onto the stack. The `G` stack frame contains three variables, `arg1`, `arg2`, and `b`, and `G` points back to the `F` stack frame that created it.

Once `g()` "returns", it sets the `ret` value in `F` so that way `f()` can have access to the value. Then it exits and the stackframe `G` is popped off the stack, and control returns to the `F` frame (where it last left off, now with the return value). Then once `F` is done processing, `F` is popped of the Stack and only `main` remains.

This diagram shows the stack after `main` has called `f` and `f` has called `g`. Notice how we draw `f` below `main` and `g` below `f`. This is because the stack grows down.



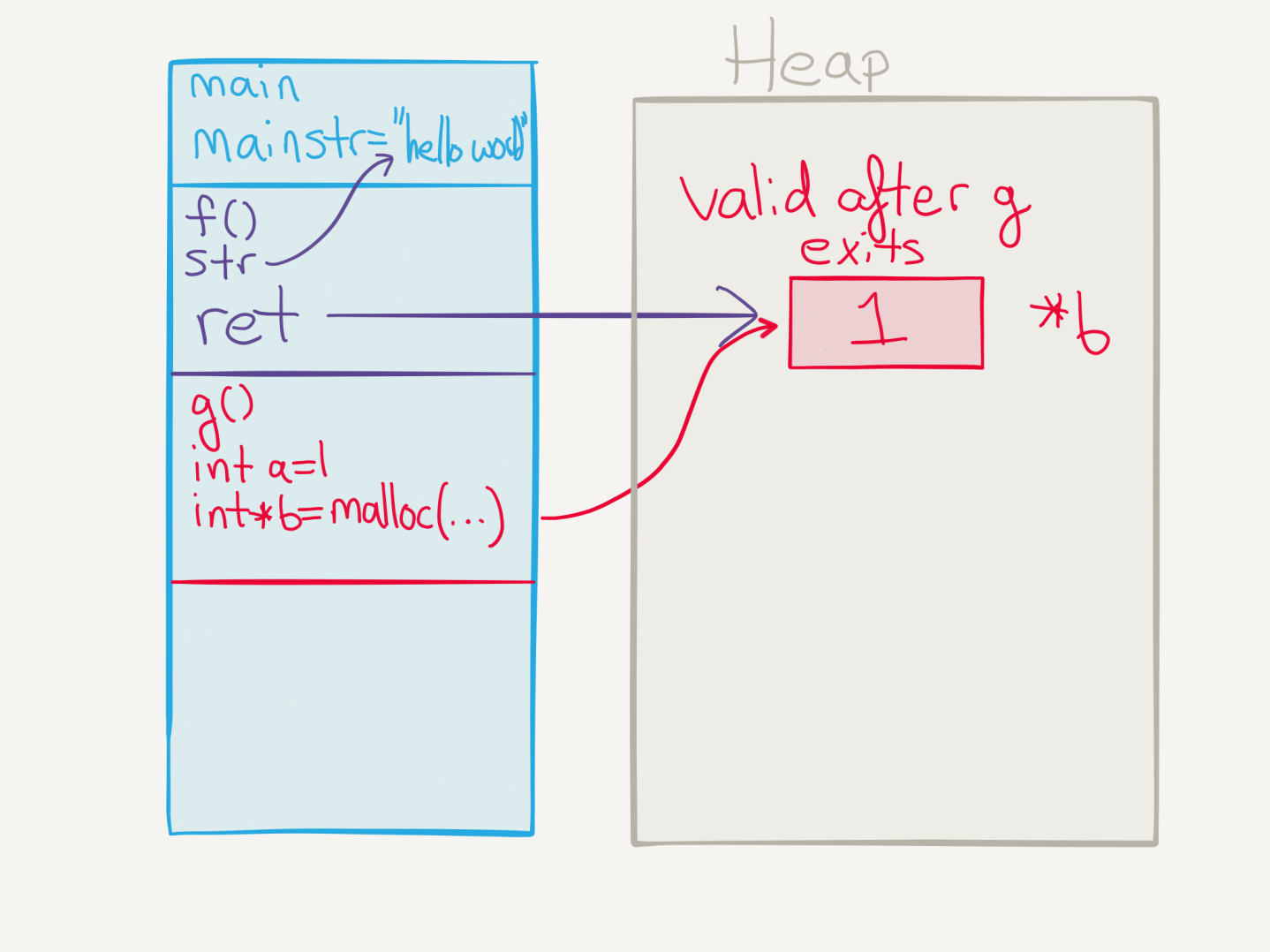
This is why you cannot return pointers to local variables. In this case if greturned a pointer to b,then once greturns to f,Gwould be popped off the stack and bwould no longer exist. Any attempt by `f` to use the pointer to `b` would be invalid because `b` doesn't exist anymore! It has been popped off the stack. If you had returned a pointer to it, it would be called a "dangling pointer"! Since each local variable only lives as long as the stack frame of the function it is in, a pointer to a local variable is only valid for "children" of this function (like `g` is a child of `f`) but it is invalid for parents of functions (like `f` is a parent of `g`). This means you should never return a pointer to a local variable, but it is fine to pass to functions pointers to your local variables.



Heap

If we could never return pointers to values created in those functions then we would be in big trouble. Let's say we wanted to create a `Vertex` struct, we could never create a function called `newVertex` which would return a pointer to a new `Vertex`. Or we could never write a function that returns a string that is created by the function. This would be a big problem.

This is why we have the notion of a heap. It is memory that is specifically set aside from the stack where we can "allocate" objects. The Heap is essentially a subset of the `Memory` array that for our purposes we can call `Heap`. We are guaranteed that Stack will never use the memory in `Heap` though elements on the Stack can point to elements on the Heap.



That said, using if we were to use `Heap` (as an array), directly, it would be quite overwhelming. What elements have I allocated? How can I be sure that I am not using this memory for something else? For this C offers us a memory manager. This memory manager comes in the form of 2 simple functions `malloc`and `free`, as well as some other simple library functions that build on top of these.

Malloc

Malloc is a function that essentially exposes the interface: Give me a pointer to an unused block of memory of size `s` on the Heap. And that is all `malloc` does, you call `p = malloc(s);` and it will give you back a pointer `p` to a block of memory on the heap of size `s`. This memory is safe to return from a function because `Heap`is global, and will still be accessible after the function returns.

Free

Free is a function that exposes the interface: I am done using this pointer, you can use the memory for something else. You just call `free(p)`.But, once you call `free(p)`you should never use `p` again: you should never call `*p`again and you should never `free(p)`again.

If you ever use `p`again you will get a `use after free`error if you are running under valgrind. And if you call `free(p)`again you will get a `double free`error.

Realloc

Realloc exposes the interface: resize this pointer `p`to a new size. Realloc is essentially just the following function:

```
void* realloc(void *p, int size) {
  void* old = p;
  p = malloc(size);
  memcpy(p, old, sizeof(old));
```

```
  free(old);
  return p;
}
```

Notice: This function invalidates your old pointer by freeing it! PS: There are bugs in this function, so don't use it directly, but it gives the essence of what realloc does.

This means that if there are two functions `f()` and `g(int *array)`:

```
void f() {
  int size = 10;
  int *array = malloc(size*sizeof(int));
  g(array);
  array[0] = 1; // ERROR: array is no longer be valid
}

void g(int *array) {
  array = realloc(array, 20);
}
```

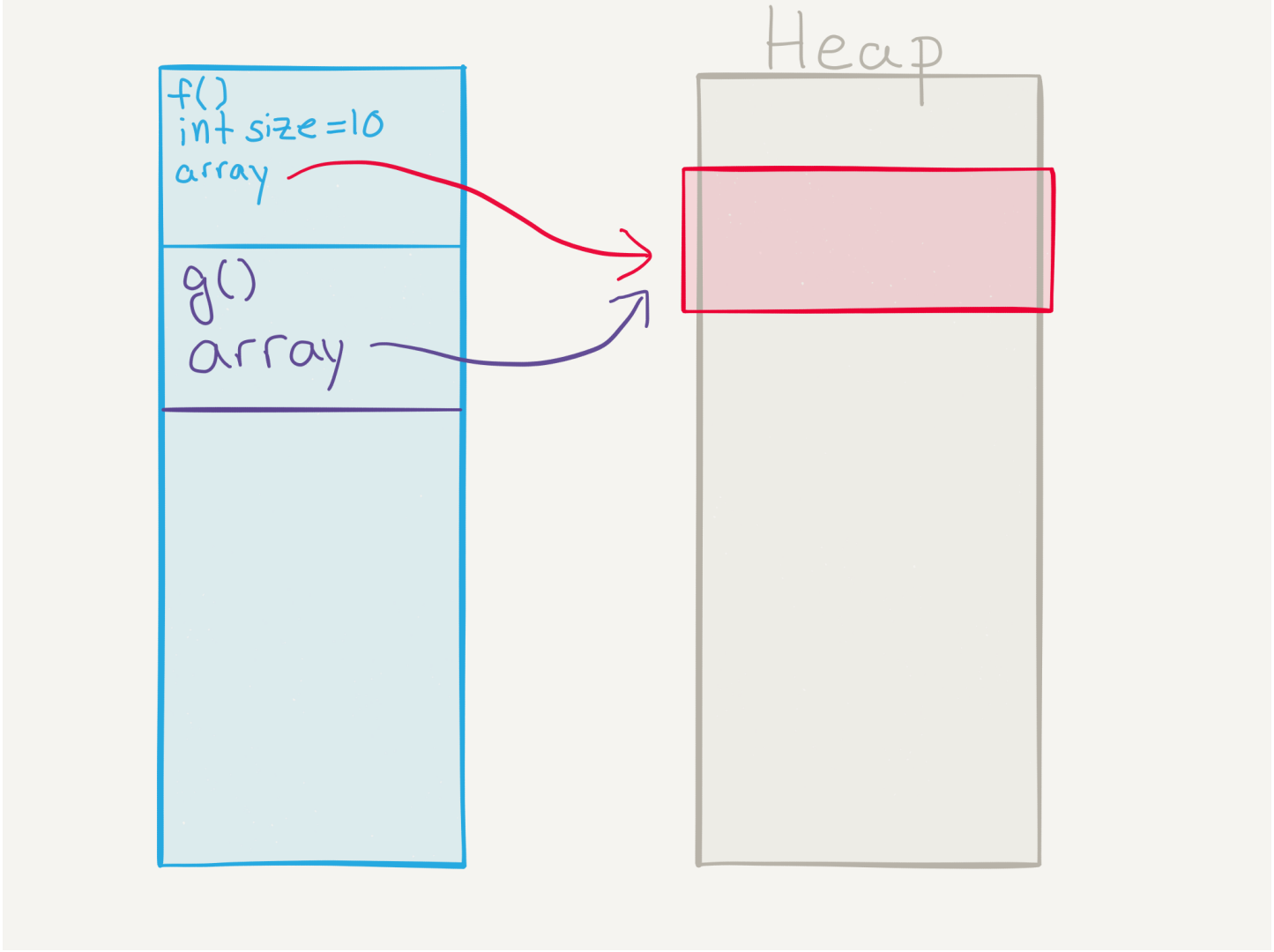
This code has an error because `g` has freed `f`'s array. Remember `realloc` might free the pointer it is resizing.

This code can be fixed by passing a pointer to array, rather than the array itself.

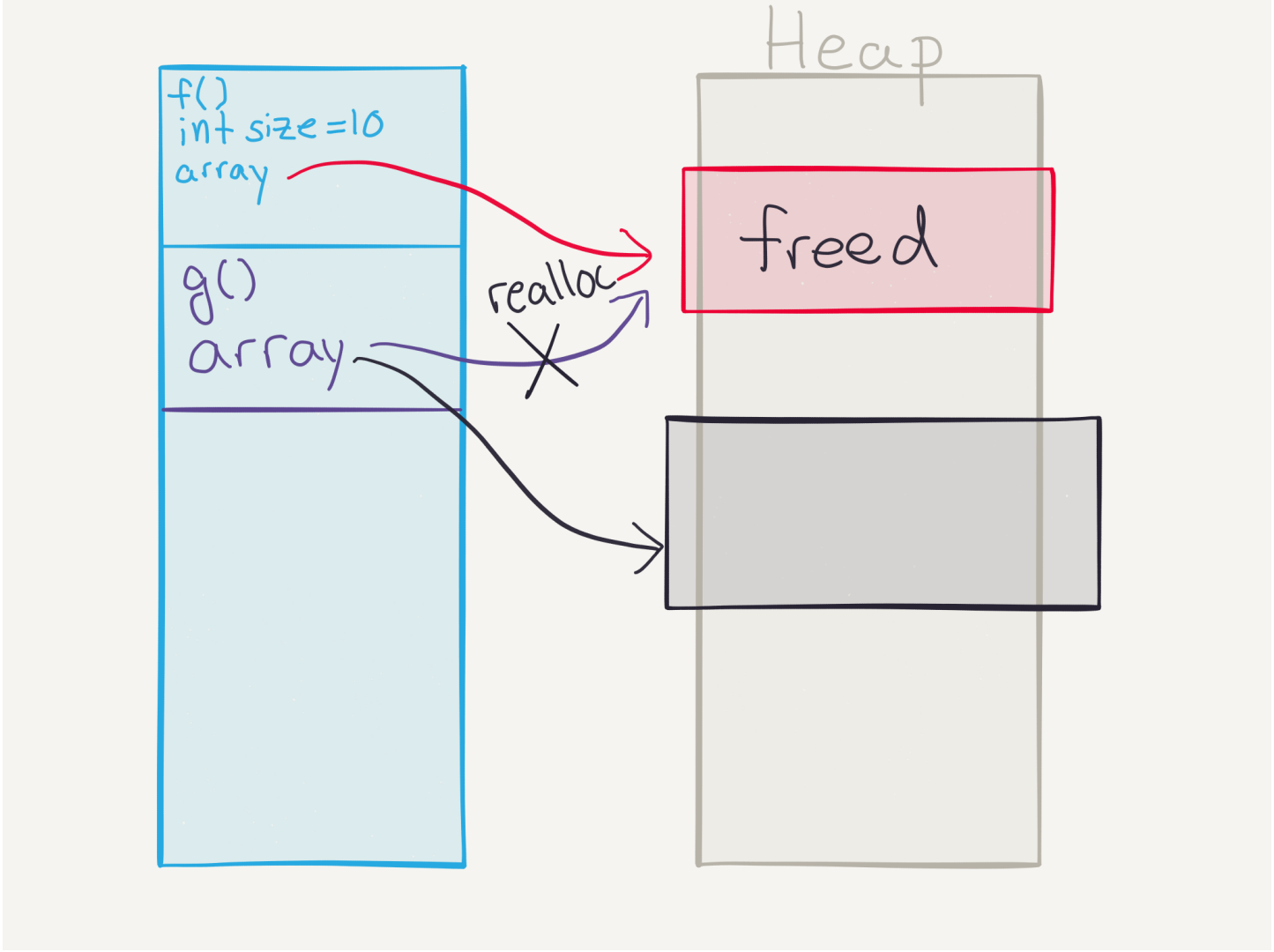
```
void f() {
  int size = 10;
  int *array = malloc(size*sizeof(int));
  g(&array);
  array[0] = 1; // GOOD: array has been updated
}

void g(int **parray) {
  *parray = realloc(*parray, 20);
}
```

As we discussed in the previous sections, pointers are just integers. The problem with the first code listing is that array is at some memory location `i`,but then when it is realloced it is moved to memory location `j`. The function `f` passed `i` to `g`. And then when `g` returned `f`'s ponter `i` was now invalidated (since the memory is now located at `j`).

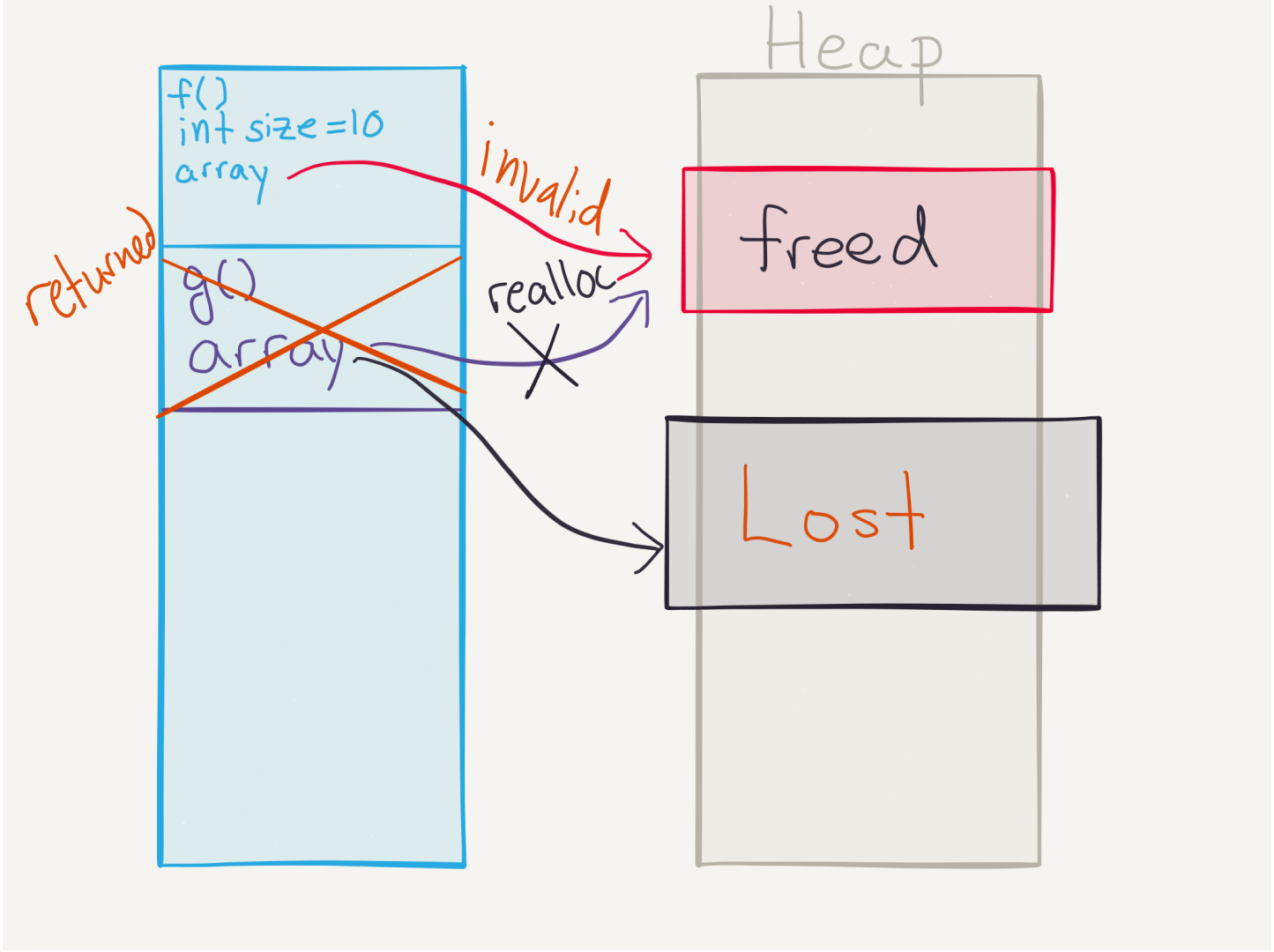
A step by step breakdown of how this problem appears: First `f` allocates an array (`array`) on the heap, and passes a pointer to this array to `g`.



Next `g` reallocs its pointer to `array` thus freeing the old array, and creating a new `array`. Notice `f's` pointer `array` is no longer valid as it still points to the old block of memory that has now been freed.



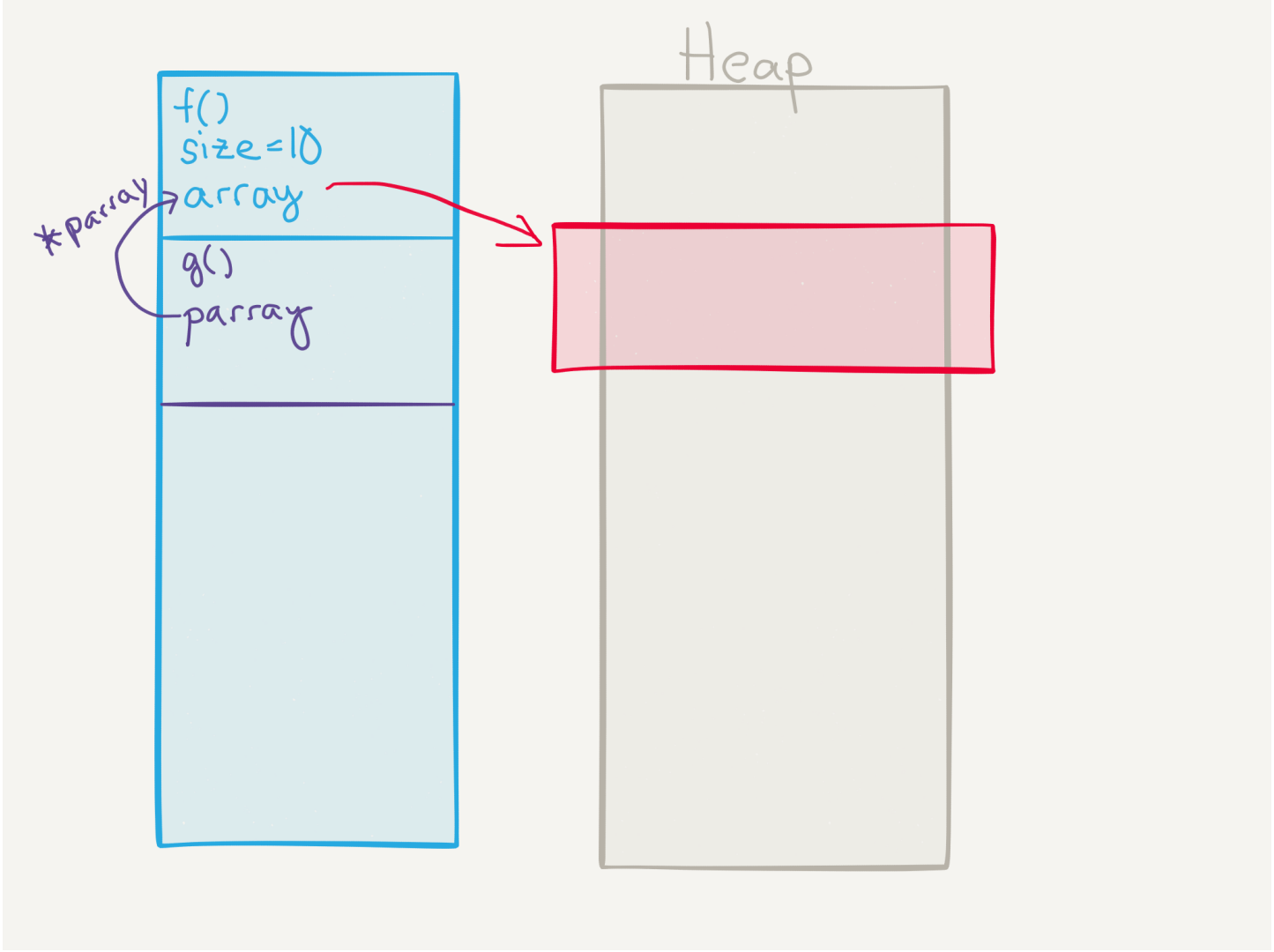
Now `g` returns to `f` and `f's` `array` pointer still points to the invalid block. Furthermore the new `array` that was allocated by `g` is now lost as we no longer have a valid pointer to it.



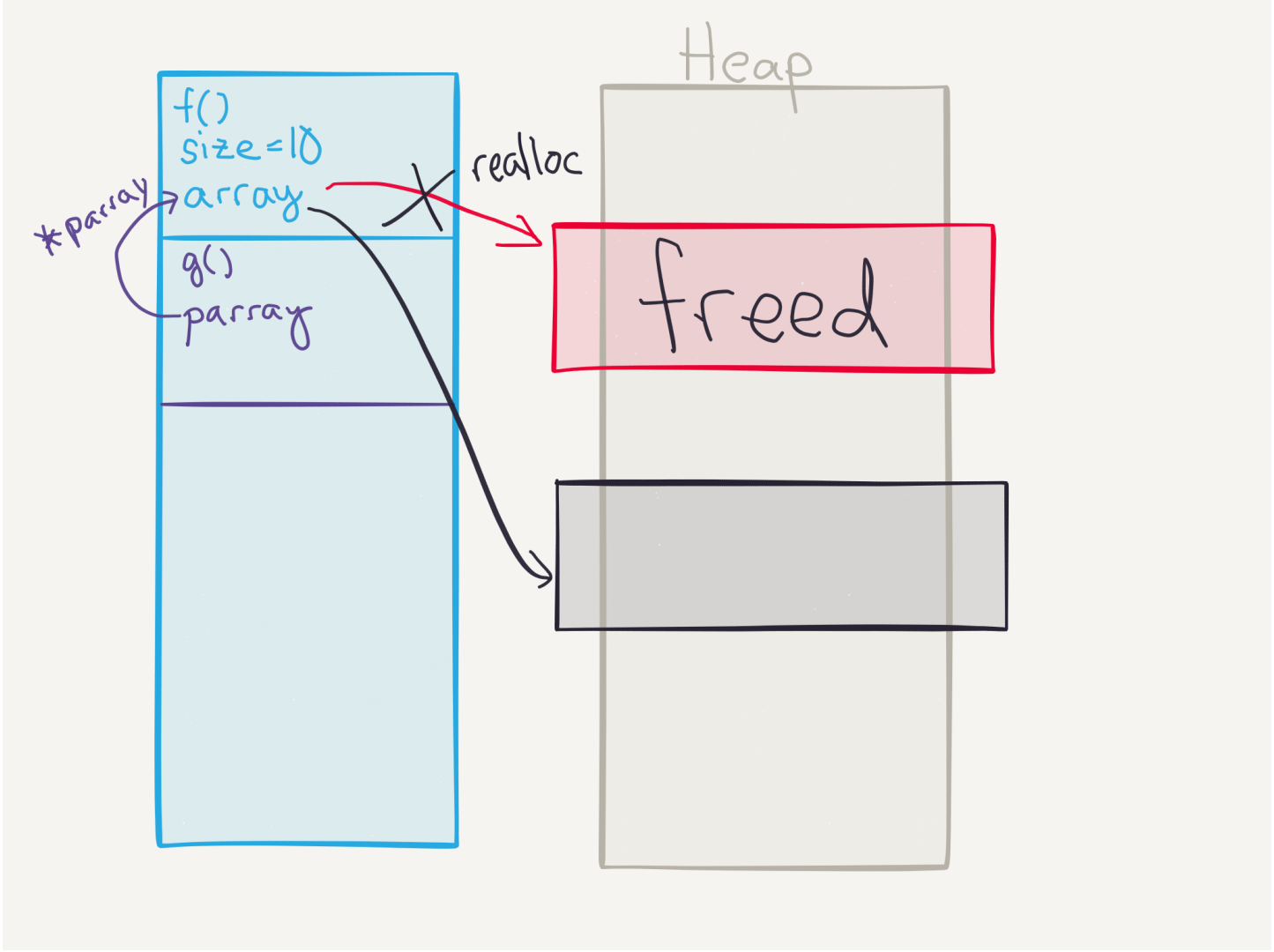
This second code listing fixes this problem by passing a pointer to `i` (`parray=&array`) instead of `i` itself. This means that when `g` sets `*parray = realloc(*parray, 20)`,the pointer `array`is updated to point to this new location. The result of this is that `f`now knows that `array`changed locations. Essentially we give `g` access to `f's` local variable `array`, to make any changes that are needed to it.

By adding an extra level of indirection we are able to keep track of changes made to the pointer. Essentially if we want to keep track of a value that changes we need to have a pointer to it. In this case, since the value that changes is a pointer (`array`) we actually need to have a pointer to that `array`.This is why we need to pass in a `int **parray`.

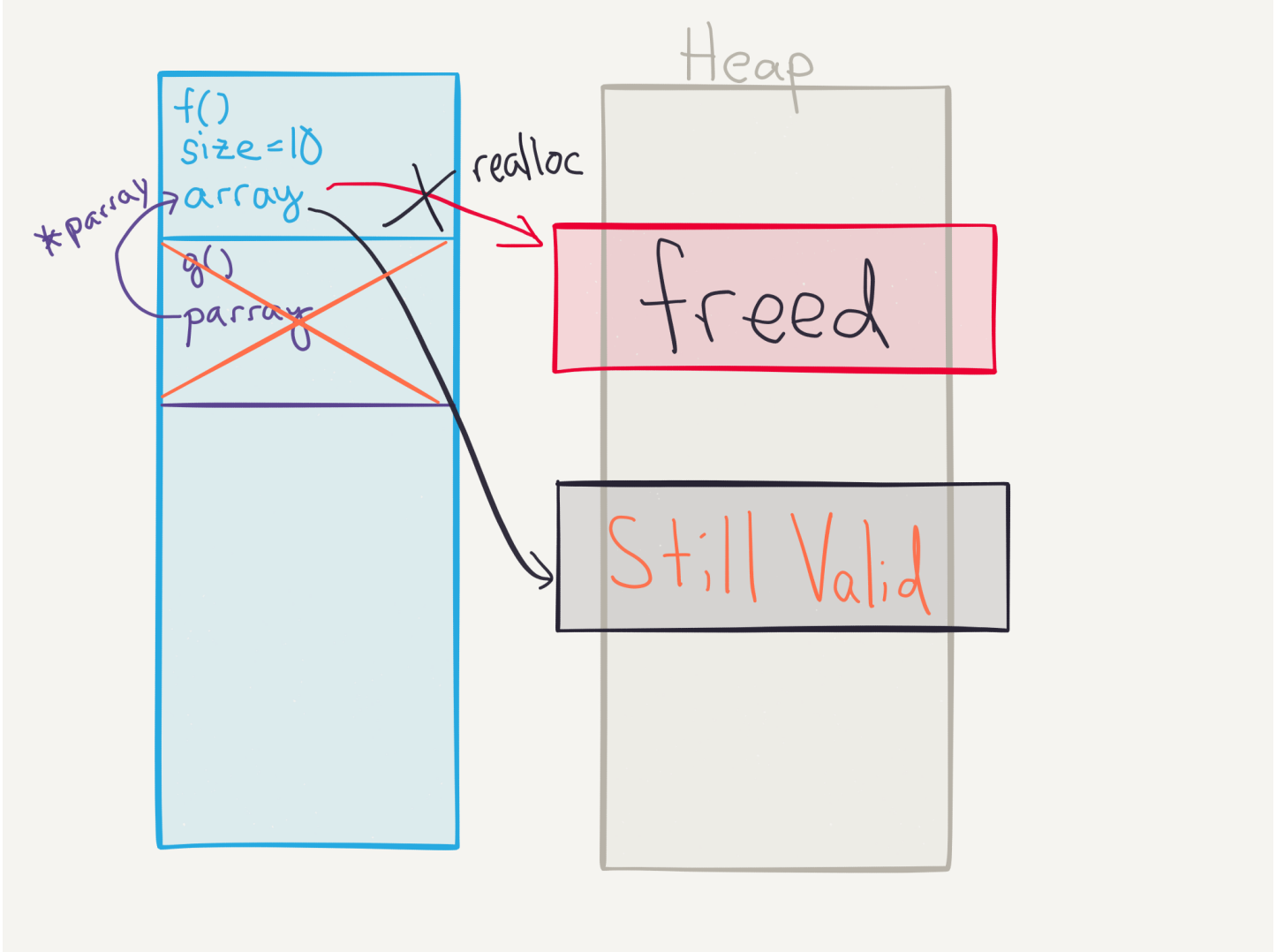
The starting state is almost exactly the same as before except for with one main difference. `array` is still allocated on the heap, however, `f` passes a pointer to its `array`, local variable, to `g` instead of a pointer directly to the data in the array.



This means that when `g` reallocs `*parray`, the `array` variable `f` is now updated to point to the new array as well. The old memory behind `array` was freed and there is nothing pointing to it.



Now when `g` returns `f` is in a valid state. It's array pointer is pointing to the still valid block that `g` created by reallocing `*parray`.



Designing Memory Safe Code

Given what we know about memory safety in C, there are a few tricks that we can employ to help us develop memory safe code. These methods are not dogma but rather ideas that I have found quite helpful when I write code in language with manual memory management.

Ownership

The first, and probably the biggest thing, that can help you write safe C code is by maintaining the idea of ownership. All data in your program should have a clear owner. Ideally you write comments in your code to indicate where ownership is transferred.

All your variables should have a clear owner. When you pass a pointer `p` to a function `g` you have transferred ownership to `g`. Unless `g` does one of the following:

- Returns an updated (valid) pointer `p` as a return value.
- Takes a pointer to your pointer `pp = &p` and only mutates (`*pp`).
- Guarantees that `p` is not mutated (though *p can be).

Then `g` consumes `p` and it should not be used after `g` returns. If ownership is passed back, our value is effectively “borrowed” by `g`. When writing code you should always mark functions that consume a value with a comment. This way, when you call these function, you know you should not use your value after. Whenever a value is “borrowed” by a function, always make sure that that function follows one of the three rules specified above, and above all make sure that your pointers are still valid after the function returns.

Though maintaining the idea of ownership might seem complex, it is really just formalizing a process that you already probably do. Otherwise how can you know whether `f` can use `p` after calling `g(p)`? It could be that gtransferred ownership to a structure, and now your function `f` is accidentally mutating some variable that is also shared by your structure. This

behavior is called aliasing and though aliasing is not illegal it does make it significantly harder to reason about the state of your program.

Lifetimes

One of the big advantages that ownership gives us is that we can now start to reason about the lifetime of a variable. Lets just say that you:

- In function `f`: malloc some `memory`
- Give a pointer to this `memory` to a structure `S`
- Do some operations on `S` which causes `memory`to be reallocated
- Access `memory` in `f's` local variable.

This is the same as the invalid code mentioned in the heap section. This is a use after free error. This happened because reallocing `memory`caused my old value to be freed, but my function `f`still has a pointer to the old value and tries to use it (even though it has been freed). This is a big error in a program that displays how aliasing can make code harder to reason about. Specifically in this case, because there are two different entities `f`and `S`which think they own `memory`.This means that when one reallocs `memory`the other won't know about it. Therefore we don't know about the lifetime of this variable `memory`. In this case both `f` and `S` are sharing a pointer (`memory`) to some data. When you want to share a value like this (just like between `f` and `g` in the fixed heap example) you pass a pointer to what you are sharing (`&memory`).

Sharing through a proxy

The reason why this works is because you are sharing a pointer (`pp == &memory`) which has a known lifetime (and thus location). This value acts as a proxy for accessing `memory`. By using something (`pp`) that always exists in the same place, we can access something (`memory`) that changes locations.`pp` will always be in the same location (it is just an integer), while `memory` might change locations when it is realloced. Since `pp` proxies all accesses to `memory` it will ensure that you are always access the right variable. This is because `pp` always will point to a valid memory block, since you will do `*pp = realloc(*pp, new_size)`.

This way of viewing and creating memory safe code combines a few concepts from the ownership section. Specifically `pp` is the sole owner of `memory`. This is because you can only access `memory` through `pp`. Therefore there is only one way to access `memory` and `memory` is always valid since it is accessed by saying `*pp`. `pp`, meanwhile, never changes location. The only thing that changes about `pp` is what `pp` points to. By using these ideas from the ownership section we are able to guarantee that the lifetime of `memory` is always long enough for both `S` and `f` to use since `pp` proxies all accesses to `memory` and `pp` is in a constant location in both `S` and `f`.

When writing code use the idea of ownership to keep a clear idea of what the lifetime of variables are. For data on the Heap this means between when it is `malloced`and `freed`.The `free` can be implicit in a `realloc`, etc. This will make it easy to always know when you can access some variable and when you should free it.

Local variables live only as long as the function they are in

For data on the stack this means knowing that a local variable `v`can only be accessed as long as this function`f`'s stack frame is valid (on the Stack). This means that `f`and any child of `f` (a function that `f`calls), or grandchild of `f`, etc. can access `v` safely. This is because these children are only alive for some subset of the time that `f` is alive and `v` is only invalidated once `f` dies. However, `main` (the function that calls `f`) cannot use pointers to local variables in `f` because they only exist while `f` is running (and therefore `main` is not).

Heap variables live until they are freed

For data on the `heap` this means if `f` mallocs a variable `p` and then free's `p`, it should only access `p` between the malloc and the free. To access it before it exists obviously doesn't make sense, and to access it after it is freed doesn't make sense.

Maintain Invariants

- Functions that allocate should always allocate. This means the caller can always free it.
- All variables returned by functions should have the same lifetime. This means if someone passes in a pointer (that points to something on the stack or the heap), you should pass something back on the heap (not conditionally back on the stack or the heap).
- Either always null-terminate or never null-terminate strings. Whatever you choose, make sure you remember that some functions expect a null terminated string (strcmp, strlen), while others don't, and instead accept a length (ex:

str**n**cmp) where the n signals that you have to specify a length.

Valgrind

You should always be running your programs with valgrind. This means that you should always at least write: `valgrind ./command`

Good tutorials for Valgrind.

Must Reads

http://zoo.cs.yale.edu/classes/cs323/doc/Valgrind

http://valgrind.org/docs/manual/mc-manual.html

Helpful

http://pages.cs.wisc.edu/~bart/537/valgrind.html

Disclaimers

Virtual Memory means we can address 2^64 on 64 bit architectures.

Stats are given assuming 64 bit architecture.

Simplifications are made. Specifically that `int` is actually a 64-bit integer. This is to simplify the discussion of pointer-integer duality.