ProgrammingParadigms-Lecture08

Instructor (Jerry Cain):Hey, everyone. Welcome. We actually have some handouts for you today. We just decided to hand them out after you all sat down. So you'll be getting three handouts and they should be posted to the website, as my TA Dan Wilson is gonna do that sometime before 11:00 a.m. – before noon. Well let's see, last time, I had just given you a little bit of an introduction as to how the heap is managed by software that's included in C libraries that are linked against all of your own code. Every time something like six degrees, or RSG or vector test, or whatever, is created – to remind you from where we were last time, we're gonna start talking about all the various segments in memory. Each application behaves as if it owns all of memory. We'll give more insight into that in a few lectures. I'm specifically interested in the segment that, in practice, is usually – I mean very approximately, but usually right around here. When an application is loaded into memory, the lowest address and the highest address of the heap are advertised to a library of code that's responsible for implementing malloc, free and realloc. Okay? Now this is a general sandbox of bytes. Malloc just hands out addresses to the internals. It actually records, behind the scenes, how big each figure actually is, so that when free is called and passed one of those three pointers there, it knows exactly how much memory to donate back to the heap.

I'll explain how that works, in a second. Because this is managed by software, malloc, free and realloc, the person implementing that can use whatever heuristics they want to make it run as quickly and efficiently and, obviously, as correctly as possible. And so we're gonna talk about some of those heuristics and how that works. Now you've always been under the impression that, when you do this – I'll just call it ARR is equal to malloc – and you do something like 40 times the size of, oops, the size of int. And actually, I'll go ahead and strongly type this pointer, to not be a void star but to be an int. You're under the impression that you get actually 16o bytes back.

I can tell you, right now, that you do not – you certainly have more than 160 bytes set aside on your behalf. If this is the heap – I won't draw all the other nodes that have been handed out, but at some point, it discovers a node that is large enough to accommodate that request right there. We think of it as perfectly sized to be 160 bytes. This address is ostensibly handed back to you and when you pass this either to realloc or to free, it actually can confirm internally – or no, I'm sorry, it can't confirm, but it just assumes that the pointer that's handed to free or realloc is one that has been previously handed back by either a call to malloc or realloc. Okay?

So that is2 some legitimate pointer that has been handed back. The way memory is set aside on your behalf is that it actually allocates more than that number. Okay? For a variety of reasons. But what it will normally do before it hands back an address to you, if you ask for 160 bytes, it'll usually set aside 164 bytes or 168 bytes. Why? Because it'll actually include space at the beginning of either 4 or 8, or actually 16 or 32, whatever it decides, a little header on the full mode, where it can actually lay down a little bit of information about how big the node is. Does that make sense to people?

So if this really is 160 bytes and this is 4, it might, among other things, write down a 164 inside that 4-byte figure. Okay? If it's 8 bytes, it can actually write more than just the size. When you get a pointer back, you actually don't get a pointer to the head of the entire node; you get a pointer that's 4 or 8 bytes inset from the beginning. Does that make sense? You have access, seemingly, to all of that space right there. When you pass the pointer back to free – let's forget about realloc for the minute – for a moment – free says, "Oh, I'm just assuming that this a pointer that I handed back earlier. If I really handed this back, then I know that I put down, as a little footprint, how big this node was. So I'm gonna cast this pointer to be an int star or a long star, or a long long star, or whatever it wants to. So that I can gracefully back up 4 or 8 bytes, interpret those 4 or 8 bytes, in a way that I know I laid down information before." And say, "Oh look, there's a number 164." That means, from this point to that point right there, should somehow be threaded back into the heap. Does that make sense to people?

Okay. Given that right there, here are a couple of problems that I want you to understand why they don't work. Array is equal to malloc 100 times the size of int. So this is a legitimately allocated block of 100 arrays. You know that you're gonna get either 104 bytes or 108 bytes, just a little bit more – I'm sorry 404 or 408 – and then, you go ahead and you populate the array and you realize that you don't need all the space. Maybe you only need to use 60 of the integers and you've recorded that in some other variable. Okay? And so you think you're being a good memory citizen and you do this: ARR plus 60. Now the code wouldn't get written that way, it would probably be framed in terms of some local variable, say "num ints in use" or something like that, or "effective length," and you might because you're more sensitive to memory allocation and you might want to donate back what you're not using, might think that that should work. Well, if this is the 400 byte figure that you've logically gotten, and you've been handed that address and there's a pre-node header of meaningful information there, meaningful to malloc and realloc, and you go ahead and hand back that address to free, depending on the implementation it may be very naïve and just assume, without error-checking, that it was something that was handed back via malloc or realloc before. It might do some internal checking, but malloc and free realloc are supposed to be implemented as quickly as possible and not do the error checking for you because they're assuming that you're really good at this C and C+ + programming thing, So they're not gonna interfere with execution by doing all of this error checking every single time free and malloc get called. So if you were to do this, right here, and it doesn't do any integrity checks on the number that it gets, it will blindly back up 4 or 8 bytes, whatever happens to reside in the two integers, or the one integer at index 58 and 59, would be interpreted as one of these things right here. And if it happens to store in the place where 164 is stored, right there, if it happens to store the number 29,000, it will go from this point forward 29,000 bytes and just blindly follow whatever algorithm it does to integrate a 29,000 byte block back into the heap. Okay?

Does that make sense? Now you're not gonna say, "Sure, what impact that has on the heap and what data structures look like." I'll give you a sense in a second. But the bottom line, the takeaway point here, is that you can't do that and now you have some insight as to why. Okay?

If you do this, int array 100, and you statically allocate your array, and you don't involve the heap at all, and you use it and because you're new to C programming and you think that you have to free the array, if it doesn't do any error-checking on the address, it's not even obligated to do any error-checking to confirm that it's in the heap segment in the first place, it would go to the base address of your static array, back up 4 or 8 bytes, whatever figure and bit pattern happens to reside there would be interpreted as the size of some node, okay, that should be incorporated into the free list data structure. Okay, I'm sorry, I shouldn't say free list because you don't know what that is yet.

Incorporated into the collection of free nodes that the heap can consider for future – in response to future call to malloc and realloc. Okay? Is this sitting well with everybody? Okay? The best implementations – or well, best is up for debate – but the fastest implementation is: Don't do error checking. They'll use heuristics to make sure they run as quickly as possible. Some implementations, I've never seen one, but I've read that some implementations do actually basically keep track of a set of void stars that have been handed back.

And it'll do a very quick check of the void star, that it gets passed to it, against – and make sure it's a member of the set of void stars that have been handed out. And if it's in debug mode, it might give an error if it's not present. If it's not in debug mode, it may ignore it and say, "I'm not going to free this thing because it will only cause problems later on." Okay? Does that make sense to people? Yes, no? Got a nod. Okay.

Now, as far as this 160 is concerned, that is not exactly a – that's not exactly a perfect power of 2. Implementations that I've seen, if this is the entire heap, I've seen this as a heuristic. When you pass in a numbytes figure to malloc, let's say that it's 6, if you quickly recognize whether or not the number that's supplied is less than, say, 2 to the 3rd or 2 to the 5th, or 2 to the 7th, basically categorize and throw it in a size bucket as to whether it's small, medium or large. I've seen some implementations actually divide the heap up, so that anything less than or equal to, say 2 to the 3rd equal to 8 bytes, is allocated from right there. Anything less than or equal to 2 to the 3rd, I'm sorry, 2 to the, like, 6th equal to 64, might be allocated from this segment. And it would always give out a block that is, in fact, exactly 64 bytes or 8 bytes long. Okay? In other words, it won't try to actually perfectly size everything because that takes a lot of work. If it can take some normative approach as to how it clips off individual segments within each sub segment, it might actually have a much easier time keeping everything clean and organized. Okay? As long as it allocates – if you ask for 160 bytes, if it goes ahead and allocates 192 bytes, or 256 bytes, you actually don't mind all that much because at least you're given – you're certainly given enough memory to meet the 160 request. Okay? Does that make sense? Yeah?

Student: That doesn't necessarily mean that, if you have an array of 160 bytes, you [inaudible]

Instructor (Jerry Cain):Right, you're not supposed to rely on implementation details because the implementation of malloc, free, and realloc on, say, you know, one flavor of

Linux may actually be different than it is on another flavor of Linux. I mean, probably not. I'd say all of those compilers are probably GNU backed, so it's like GCC. But like, for instance, Code Warrior vs. X code on the Macintosh. They are implemented mostly, I mean primarily, by two different sets of engineers. And one may use one different heuristic for how to allocate stuff and those who wrote X code may have gone with a different approach.

So you certainly can't assume you have that memory. You're just supposed to assume that you've got the 160 bytes and that was it. Okay? Do you understand, now, a little bit why running over the boundaries of an array can cause problems? Sometimes, it doesn't. The most common overrun, actually, is at the end of the array. So you'll do something like i less than or equal to 10 as opposed to i less than 10. You'll write one space too far. Consider the less common but certainly not unrealistic situation where you actually visit all your elements from top to bottom, you go one element too far, and you actually access and write to array of negative 1. You know where that resides. It happens to overlay the space where malloc and realloc actually put down information about how big the node is. So if you, for whatever reason, go and zero out from 100 down through zero, well then you actually go 1 too far then you zero out the 1 byte – the 4 bytes where malloc is really relying on meaningful information to be preserved. And it will completely toy with the implementation of malloc and realloc in its ability to do the job properly. Okay? Does that make sense? Okay.

What else did I want to talk about? As far as how it keeps track of all of the different portions within the heap, that are available to be handed back to the client, in response to malloc, realloc calls, let me go with this picture. Let's assume that this is the entire heap and I'm writing it to emphasize that it's a stream of bytes. And a snapshot of any one moment – this is allocated out, right here, and this is allocated out, and let's say that this is allocated out, as well. Each of these will probably have a pre-node header at the front. Okay? With meaningful information – they can actually keep more than just the size of the node.

In fact, some implementations – I have seen this, will not only keep track of how big this node is, but it'll actually also keep track of whether or not the node after it is free or not. So it can kind of simplify or optimize the implementation of realloc. So it can keep a pointer to this right here. Does that make sense to people? Okay? And it can go right here and see what – how big it is and whether or not it can accommodate the stretch that is an option in response to a call to realloc. As far as all these blank nodes are concerned, it wants to be able to quickly scan the heap for unused blocks whenever malloc and realloc are called. What will typically happen is that it will interpret these nodes right here as variably sized nodes. You're familiar with that from Assignment Two and it'll overlay a link list of blobs over these 3 ravens right here. So the beginning of the heap is right there, right at the beginning, so it knows where the next pointer is. It'll use the first 4 bytes of the unused blob to store the address of the next blob. Okay? It'll store the address of the next blob right there, and then maybe, it'll put null there or maybe it'll cycle back to the front and use something of a circular link list approach. Every single time you call malloc or free, it obviously wants to traverse this link list and come up with

some node that's big enough to meet the request. More often than not, I see a heuristic in place that just, basically, selects the first node that it can find that actually meets the allocation request. So if this type of thing isn't in use and it's not segmented down into sub segments, and it's just one big heap, it might start here. And if it's just looking for a figure big enough to accommodate 8 bytes, maybe this one will work. Okay?

If it needs 64 bytes and this is only 32, but this is 128, it will say, "You're not big enough, but you are." Does that make sense? So it would approach this first fit heuristic in searching from the beginning. There are other heuristics that are in place. Sometimes, they do aggressively search the entire heaps free list. That's what this thing is called. That's what I used earlier. And it'll actually do an exhaustive search because it wants to find the best fit. It wants to find the node that is closest in size to the actual size that's needed by the call to malloc so that as little as memory as possible is left over in some free node. Does that make sense? Okay?

I've seen – I've read about, although I've never seen, that sometimes they use a worst fit strategy. Which means they basically scan the entire heap for the biggest node and use that one with the idea that the part that's left over will be still fairly big. And we're not likely to get little clips of 4 and 8 bytes which are gonna be more or less useless for most malloc calls. Does that make sense?

I have seen heuristics where they actually remember where they left off, at the end of malloc. And the next call to malloc or realloc – I'm sorry, the next call to malloc will actually continue from that point. So that, all parts of the heap are equally visited during an executable that runs for more than a few seconds. Okay? So you don't actually get lots of complexity over here and then this unused heap over here. Okay? Does that sit well with everybody?

There are all types of things that can be done here. There's extra meta-information can be stored here, and there, and there, about what comes afterwards, so it can simplify the implementation of free and realloc. If I go ahead and free this node right here, when it's actually freed and donated back to the heap, none of this is changed, but the first 4 bytes actually set to thread to that right there. And this right here would be updated to point to there instead. Does that make sense to people? Okay? It wouldn't actually go out and clear any information because it just doesn't want to bother doing that. It could, but it's just time consuming. And this could, in theory, be 1 megabyte and why go through and zero out 1 megabyte of information, when the client's not supposed to touch it again? Okay?

Let me just erase this, not because it's would be cleared out, but just so it looks like a free node, like all the other ones. Some implementations would actually prefer not two so-and-so size nodes together but one big node, since they can't see many disadvantages to having one very large node vs. two side-by-side smaller nodes. Does that make sense?

So some of them will go to the effort of actually coalescing nodes so that the free list is simpler and they have a little bit more flexibility as to how they chop things up. Okay?

Make sense? I have seen, and this is kind of funny, I have seen some implementations of free actually just record the address as something that should be freed, ultimately. But it actually doesn't commit to the free call until the very next malloc call or, in some cases, until it actually does need to start donating memory back to the free list because it can't otherwise meet the request of malloc and free alloc – malloc and realloc. Does that make sense?

So I'm speaking in, like, run-on paragraph form here about all the things that can be done. That's because it's written in software and people, those that design these things are typically very good systems programmers. They can adopt whatever heuristic they want to, to make the thing run, certainly correctly but as efficiently and elegantly as possible. Okay? And if, ultimately, this all starts out as one big random blob of bytes, the way the heap is set up is that the address of the entire free list is right there and the first 4 bytes have a null inside of it. Okay?

And that basically means that there's no node following this one. It would also probably have some information about how big the entire heap is. Okay? Because that's just the type of information that's maintained on behalf of all nodes at all times, so it just happens to be the size of the heap when things start out. Okay?

Now, consider this problem right here. Here's the heap again and let's say the entire thing is 200 bytes wide. This is free and it's 40 bytes. This is allocated, and let's say it is 20 bytes, not drawn to scale. Let's say that this is 100 bytes wide and it is free. Let's say that this is in use; it is 40 bytes wide. And is that gonna work out? No, let's make this a little bit smaller. Let's make this 80; make this 20 and so this, right here, is unused. And of course, the heap is much bigger than this and the way it's been chopped down in used block and unused blocks, is a little bit more complicated than this. But you certainly understand what I mean, when I say that there are 160 free bytes in my heap of, that's normally of size 200. Okay? If I go ahead and make a malloc request for 40, it could use this because it's the best fit or the first fit. It could use this because it's the worst fit. Okay? It can use whatever it wants to as long as it gets the job done and you're not blocked by a faulty implementation. If I ask for 100 bytes, it's not gonna work out, right? Because, even though the sum of these free nodes is a whopping 160 bytes, they're not uniformly aggregated in a way that you need when you malloc 100 bytes or 160 bytes. Okay? I mean, you really have to assume these things are gonna be used as an array of ints or structs, or whatever. And you can't expect the client to really absorb a collection of pointers and figure out how to maintain an array or overlay an array over multiple blocks. So you may ask yourself, "Well, can I actually slide this over and slide this over even further, to come up with this type of scenario where this at the front and this is right next to it? And then I really do get 160 bytes that are free – 20 and 20, so that if I ask for 100 bytes, I can actually use it?" This process, it does exist. It won't exist in this problem for – in this scenario for reasons I'll explain in a second. But what I'm really doing here is I'm just basically compacting the heap like a trash compactor compacts trash. Okay? Try and get it as close to the front as possible, okay because you know that, what remains after everything's been sifted to the front, is one very large block. Does that make sense? Okay? That's all fine and dandy except for the fact that you, very likely, I'm sure it's the

case, have handed that address and that address out to the client. And it's going to resent it if move the data out of its way. Okay? For you to bring these over here means that the client is still pointing there and right there and now, they have no idea where their data went. Okay? That's a problematic implementation.

So there are some benefits of actually compacting this heap. Okay? Now I haven't seen any modern systems do this, but I know that the Macintosh did this about 12 years ago, when I was doing more systems work on the Macintosh. Oops, you know that's fun. Recognizing that there are some advantages to being able to compact the heap to create fewer large nodes as opposed to lots and little fragmented nodes, they might clip off some part of the heap. This is used in a traditional manner by the heap manager. It's directly dealt with by malloc, free and realloc. But they might clip this part off and, even though it looks small on the board, it would in theory be a pretty large fraction of the entire heap, which is, you know, megabytes or – megabytes or gigabytes in size. It would manage this via handles. So rather than handing out a direct pointer into the heap, like we have up there, and we saw that interfered with heap compaction, what some operating systems did, in addition to malloc and free and realloc, would actually hand out what are called handles, which are not single pointers directly to data, but pointers that are two hops away as opposed to one hop away from the data. And you may say, "Well, why would they want to do that?" Well, they would not only hand out double pointers, but they would actually maintain a list of single pointers. If you ask for 80 bytes via a handle as opposed to a pointer, then it could maintain a table of master pointers and hand out the address of that to the client. Okay? So the client is now two hops away from his data, but the advantage is that this is owned by the heap manager and, if it wants to do compaction on this upper fourth of the picture, it can do it because it can also update these without affecting your void double stars. Does that make sense? Okay?

I saw that used in Mac OS 7.6 and Mac OS, like 8 – not actually 8 never existed, but like all of the 7 series of Macintosh Operating System, probably from 1994 and '95, when I was doing some consulting work that involved this. Very clever idea, the problem is that thing is compacted in the background, at a low-priority thread. Okay? Or it becomes higher priority if there's a demand for this and there's no space for it. You can't actually have the heap compaction going on simultaneously to an actual double d reference request because you can't actually have data sliding in the moment while you're trying to access it.

So the paradigm, or the idiom rather, that people would use for this is, if you really wanted to get memory to something that's compacted and managed more aggressively by the heap manager, you would do something like this. Void star star handle and it would be some function like new handle. That's what it was in Mac OS 7.6 days. You might ask for 40 bytes. Okay? Recognizing that the 40 bytes might be sliding around in a low-priority thread in the background, to keep that thing as compact as possible, you wouldn't bother with it when you knew that you were gonna certainly read to but even read from those 40 bytes. That you actually, somehow, had to tell the operating system to stop moving things around, just long enough for me to read and/or write. Okay? So you would do something like this, handle lock, which basically puts safety pins on all the blocks in

that upper fourth of the diagram and you say, “You can move everything else around, but you better not change the pointer that’s maintained in this table, that’s addressed by my handle. Because I’m gonna be annoyed if you do because I’m about to manipulate it right here.” And if you’re a good programmer, when you’re done, you unlock, you call handle unlock, so that flexibility has been restored for the heap manager to start moving things around, including the block that’s addressed right there. Does that make sense? So there’s a hint of, like, concurrency there. We’ll talk about that more in a few weeks. But that’s how heap compaction, as an idea, can be supported by a heap manager, okay, without interfering with your ability to actually get to the data. Okay? You never really lose track of it because you’re always two hops away as opposed to one hop away. Okay? Does that make sense? Okay, very good. I’m trying to think what else. I think that’s enough. Just kind of a hodgepodge of ideas. I’m not gonna have you implement any of this stuff, okay? But we’ve actually – I’ll try and dig up a section problem, not for this week but for the week after, where people on the mid-term implemented a little miniature version of malloc, where you actually had to understand all of this stuff. Okay, now I can tell you right now that I’m not gonna do that because that problem didn’t go very well, when I saw it was given. But it’s certainly a suitable section problem and something that you can do in like a less time-pressured environment. Okay? Make sense? Okay. Software managed. It is managed entirely by malloc, realloc and free. What I want to do now, is I want to start talking a little bit about the stack segment. Let me get some clear board space here. The first 20 minutes of this actually very easy to take. It’s Monday that’ll be a little bit more intense. When I talk about the stacks thing, I drew it last time; it’s typically at a higher address space, as a segment. This is the entire, as a rectangle, the entire thing is set aside as a stack segment, but you’re not always using all of at any one moment. In fact, you’re using very little of it when a program begins because there’s so few active functions. Okay? Usually, the portion that’s in use is roughly – it’s very rough –roughly proportional to the number of active functions, of the stack depth of all the functions that are currently executing. Okay? I will draw more elaborate pictures within this in a little bit. But let me just invent a function and I don’t care about code so much as I just care about the local variable set. Let’s say that the main function – let’s not worry about any local, any parameters – let’s say that it declares an int called “a.” Let’s say that it declares a short array called “b” of length 4, and let’s say that it declares a double called “c.” Okay? Actually, I don’t want to call this “main.” Let’s keep it nice and simple; let’s just call it the “a function.” All of the implementation of “a” does, is it calls “b” – I’m not concerned about passing parameters, and I’d call “c” afterwards, and then it returns. We’ll blow this off for a second. When you call the “a” function, obviously space needs to be set aside for this int and those 4 shorts and that single double there. 1-byte figure, four 2-byte figures and one 8 byte figure. Not surprisingly, it’s somewhat organized in the way that it clips off memory. It doesn’t take them from the heap; it actually draws the memory from the stack. All of the memory that’s needed for these four variable right here – I’m sorry, three variables and technically, like six because there’s four of them right here, they’re packed as aggressively as possible and there’s even some ordering scheme that’s in place. Because “a” is declared first, “a” gets a 4-byte rectangle, and because “b’s” declaration is right afterwards, it’s allocated right below it in memory. Okay? This being an 8 byte double would have a rectangle that’s twice as tall and I’ll just do that to

emphasize the fact that it's two [inaudible] as opposed to one. Okay? Does that make sense?

When you call this "a" function, what happens is, if I go ahead and just represent – I can't – I don't want to draw this entire thing again. So just let this little picture right here be abbreviated by the hexagon. Okay? This thing is what's called a activation record, or stack frame, okay, for this "a" function. Let's assume that "a" is about to be called. When "a" is called, this pointer, that's internal into the stack segment, it actually separates the space that's in use from the space that's not in use. It'll actually decrement this by size of hexagon. Okay? Do you understand what I mean when I say that?

And that means that the memory right here is interpreted according to this picture, right here. It's like this picture right here overlays those 20 bytes that the stack pointer was just decremented by. Okay? Notice it doesn't lose track of all the previously declared variables that are part of functions above "a" in the stack trays. And when "a" calls "b," and calls "c," it'll decrement the stack pointer even a little more. Does that make sense? Okay. Maybe element "b" – I'll keep these simple. Void "b" declares an int called "x," a car star called "y" and let's say a car star array called "z" of length 2. The stack frame would look like this. "X" would be above "y's" 4 by pointer, and then, this is an array of size 2. So this is a total of 16 bytes, there. I'll abbreviate it with a triangle. Okay? And let's say that "b" actually calls "c" as well and "c" has a very simple local variable set, I'll say a double m of length 3, and I'll just do a stand-alone int called "m" and it doesn't do anything that involves any other function calls. The activation for this would be big, there's three doubles; there's a stand-alone int below it; there's "m" – this is all of "m" of zero through 2. And I'm gonna abbreviate this with a circle. Okay?

So the very first time "a" gets called, you know just the work flow will bring you from "a" to "b" to "c", back to "b", back to "a", which will then call "c". Okay? So as far as the stack is concerned – let me just draw a really narrow stack, okay, to emphasize the fact that the full width is being used to help accommodate the stack frames right here, when "a" is called, whatever's above here, we have no idea. They're obviously gonna correspond to local variables; they're preserving their values; there's just no easy way to access them, unless you happen to have pointers passing those parameters. Okay?

When you call "a," this thing is decremented by size of hexagon. Oops. I needed to remind myself what a hexagon looked like. That's that and then, this pointer right here, is kind of the base or the entry point that grants the "a" function access to all of its local variables. It knows that the last parameter "c" is at an offset of zero from this pointer right here. Okay. 8 bytes above that is where the second to the last variable that was declared can be accessed, etc.

This pointer is actually stored on the hardware; we'll see that on Monday. But this actually what's called the stack pointer and it always keeps track and points to the most recently called functions activation record. When "a" calls "b," it decrements this even further. I'm gonna stop drawing the arrow and the triangle activation record is laid right below that. Okay? Access to this, it just isn't available to the "b" function because "b"

doesn't even know about it. It just – it's incidental that it happens to be above it, but "b" can certainly access all the variables that part of the triangle activation record. Since "b" calls "c," it's gonna lay a circle below that. When it calls "c," that circle will be alive and in use, the stack pointer will be pointing right there until "c" is done. When "c" exits, it simply raises the stack pointer back to where it was before "c" was called. Whatever information has been written there, okay, it actually stays there. Okay? But it's supposed to be out of view; it can't be accessed. Okay? So – I'm sorry, "c" returns and then "b" returns and we come back to that situation and the drawing there technically corresponds to the moment – we've just returned from "b" but we have yet to call "c." Right? And so when it calls "c," it follows the same formula. It has no idea that "b" has been called recently. And the formula here is that it just overlays a circle over the space where the triangle was before. Okay? So it actually layers over whatever information was legally interpreted and owned by the "b" function. But it doesn't know that and it's not really gonna be the case that "c" has any idea how to interpret "b's" data. It doesn't even necessarily know that "b" was called recently. All it's really doing is it's inheriting a random select – a random set of bits that it's supposed to be initialize to be meaningful if it's going to do anything useful with them. Okay? Does that make sense? Do you now see why this thing is called a stack? I'm assuming you do. Okay? Every time – before you can actually access "b's" variables, if you call "c" – "c" has to be popped off the stack the stack frame, before you can come back to "b." Okay? Now we're gonna be a less shape-oriented in a little bit and start talking about assembly code and how it manipulates the stack. So we're gonna suspend our discussion of the stack for, like half a lecture, okay? But now, I want to start focusing a little bit on assembly code. Okay? And how that actually guides execution. We're ultimately going to be hand compiling little snippets of C and C + + code to a mock assembly language. Nothing – it's not an industrial thing like MIPS or X86 or something like that. That would be more syntax than concepts. So we have this very simple syntax for emulating the ideas of an assembly language. And I want to start talking about that now. There is a segment down here. It usually doesn't have to be that big because the heap and the stack – the stack is actually never this big when it starts out; it can be expanded. The heap is usually set aside to be very big because anything that's– any huge memory resources are typically dynamically allocated while the program is running. This right here, I'm gonna refer to as the "code segment." Just like the heap and the stack, it stores a pattern of zeros and ones there, but all of the information in the code segment corresponds to the assembly code that compiled from your C and C + + programs. Does that make sense? Okay. So I'll give you an idea of what these things look like in a little bit. Let me just talk about, at least at the cs107 level, what a computer processor looks like. So I've already drawn all of RAM a couple of times, I'll do it one more time. There it is; I don't have to break it down, just stack and heap and code. That's all of RAM. In any modern computer, there's usually – now it's very often that there's more than one of these things, but we're just gonna assume a UNA processor, okay, where this is relatively slow memory. You're not used to hearing of RAM, which this is, as slow. But it's slow compared to the memory that's set aside in the form of a register set. I'm drawing it – it makes it look like it's one-fourth the size of RAM; it's not; it's much smaller. In our world, we're gonna actually assume that all processors have 32 of these registers, where a register is just a general purpose 4 byte figure that we happen to have really, really fast access to. Okay? Does that make sense to

people? I'm going to call this R 1; I'm going to call this R 2; I'm going to call this R 3 and I'm going to draw a 3 instead of a 2, etc. And those are going to be the names for these registers. The registers, themselves, are electronically in touch with all of RAM. There's a reason for that. Okay? And when I'm doing this, I'm just drawing this big network of silicone wires that are laid down microscopically – okay, not microscopically, but you know what I mean. So that, every single register can technically, draw and flush information to and from to and from RAM. Okay? The 16 or the 32 registers right here, are also electronically in touch with the electronics. It's always drawn this way; I'm not really sure why. What's called the "arithmetic logic unit" or the ALU, that is the piece of electronics that's responsible for emulating what we understand to be addition and multiplication, and left shifting and right shifting of bits, and masking it. All of the things that can be done very easily on 4 byte figures. Okay? Does that make sense to people? So we can support plus and minus, and times, and div, and mod and double less than and double greater than, and double ampersand and double vertical bar, and all of these things because of this ALU right here. Okay? It is electronically in touch with the register set. There are some architectures that use a slightly scheme right here. I'm going with an architecture or an assembly code – I'm sorry, I'm going with a processor architecture where the ALU is just in touch with these things right here, and it's not directly in touch with general RAM. The implications of that is that all meaningful mathematical operations have to actually be done using registers. Does that make sense?

Now you may think that that's kind of a liability, that you'd have to actually take something from memory, load it into a register, in order to 1 to it. Okay? Well, that actually is what happens. The alternative is for you to get this right here to electronically in touch with all of RAM and that would be either prohibitively expensive, or it would be prohibitively slow. Okay? So they can optimize on just load and store between registers and general RAM and then, once they get something into the register set, that's where they do all the interesting stuff.

So the typical idiom that is followed for most statements – anything mathematical – think i plus plus, or i plus 10 or something like that, is to load the variables from wherever they are, either in the stack or the heap, okay; load them into registers; do the mathematics; put the result in some other register, and then flush the result that's stored in the register, out to where it really belongs in memory. Okay?

Just assume, without committing to assembly code right here, that that's the space that happens to be set aside for your favorite variable, i. And it has a 7 inside of it. Okay? Let's say that somewhere, probably close by, there is a 10. Okay? And you're curious as to what happens on your behalf at the memory level in response to that right there. Well, truly, what happens is the j plus i compiles to assembly code and the assembly code is the recipe that knows how to load j and i into register set; do the addition and then flush the result back out to the same space that j occupies. Okay?

But just in terms of actually seeing where – how the 7 and 10 move around, what would probably happen is the 7 would be loaded into R 1. The 10 would be loaded into R 2. You can actually add the 10 and the 7 and store the result in R 3 because these two

registers are in touch with the electronics that are capable of doing that addition for you. And after you synthesize the result right here, you can flush it back out to j. So given what I've shown you, so far, it's not a useless metaphor for you to just think about assembly code instructions as byte shovelers, okay, to and from RAM, and also doing very atomic, simple mathematics. Okay? Plus, minus, things like that. Okay? The components of assembly code that really are different are usually implementation. I think the double e aspects of computer architecture are very difficult, probably because I've never really studied it; I'm a compute scientist. Okay? But also, the parts that are interesting to us, is how something that is stated, usually quite clearly, in C or C + + code, actually gets translated to all of these assembly code instructions, such that the assembly code, which is in touch with memory, actually executes and imitates the functionality of the C and C + + code. Okay? If this is my C + + code, okay, then I need this to become a 17. How does the assembly code actually do that for me? How do I have this one to one translation or one to many translation, between individual C and C + + statements, okay, and the assembly code instructions that have to be done in sequence in order to emulate that, right there. Does that make sense? Now, given these pictures over here, you shouldn't be surprised that j doesn't move as a variable while that program – while that statement is running. Okay? So it's always gonna be associated with the same address in RAM. Okay? That's good. You don't want it flying around when you try to write a 17 back to it. Okay? You may ask why I don't – why they wouldn't – what are the disadvantages of trying to get this to be in touch with general memory? I mean, if we have to have this in touch with that – if we have to have this complex matrix between the register set and this right here, okay, then why not just have it this complex matrix of wires between general RAM and the ALU? It just makes the implementation of the hardware that much more complicated. There's no way of getting around at least a few set of operations between RAM and the register set. Okay? At the very least, you have to have load and store instructions to move four 2 and 1 byte quantities to and from RAM, in between the register set. Does that make sense? You have to have at least those. If you actually try to support addition between two arbitrary addresses, it can technically be done. It might make a clock cycle that's actually on the order of seconds. Okay? But it technically can be done. But hardware designers obviously want the hard – want to be able to do any form of atomic operation at the hardware level, but they also want it to run as quickly as possible. So given that this right here would have to correspond to more than a few actions; load i into register; load j into a register; do the addition and write things back. There were, like basically four verbs in that sentence, four actions that needed to happen in order for that to emulate that right there. What I'm loosely saying is this is a statement that will correspond or gen – will – this will compile to four assembly code instructions, two loads, an addition and a store. If it tried to optimize and do all of this in one assembly code instruction, it could do it. Okay? It would actually probably require that this be in touch with that right there, but the clock cycle would have to increase because whenever you have a more complicated hardware implementation, it's generally the case that the clock cycle speed goes up. It may go up – if it goes up by more than a factor of 4, then you'd actually prefer the really simple, load; do the work here and then flush out. Because in the end, you're really worried about correctness but also speed. Okay? And if you have this very robust implementation, where everything can be done directly in memory, but the clock cycle is on the order of, like, milliseconds as opposed

to nanoseconds – or not nanoseconds, microseconds, okay, then you actually prefer to go with the simpler idea, this load store architecture, where you always load things into registers, manipulate them in a meaningful, mathematical way and then flush the result out to memory. Okay? So I actually don't have any time. I have 25 seconds; I can't do very much, then. What I will do on Monday is I will start seemingly inventing assembly code instructions. And I will come up with syntax for actually loading into a register a 4-byte figure from general memory, doing the opposite, taking something that's in a register and then flushing it out. And then, talking about what forms of addition and subtraction and multiplication are supported between two different registers. Okay? We'll get –

[End of Audio]

Duration: 50 minutes