ProgrammingParadigms-Lecture07

Instructor (Jerry Cain):Hey, everyone. Welcome. I have one handout for you today. It is Assignment 3. It is due next Thursday evening. You'll get Assignment 4 next Wednesday. It'll be due the following Thursday evening. You'll get a written problem set a few Wednesdays from now. It won't need to be turned in. I'll talk about that more when I actually give it out, but it'll provide a collection of written problems that you'll be responsible for for the midterm come the following Wednesday night.

When I left you on Monday, I had really just gotten through what, at the moment, was the full implementation of a generic stack. I've actually made parts of it easier than it really needs to be because we focused on storing ints, and doubles, and characters, and Booleans.

What I wanna do now is put off the implementation for about ten or 15 minutes, look at that again, and think about how we would use it to store a stack – I'm sorry, use a stack – a generic stack right there – to store a collection of strings and print them out in reverse order.

Now, the nonsense code I'm gonna put on the board is effectively gonna just print out the reverse of an array, but it's really in place to illustrate the mechanics of using those four functions when you're storing C strings. That's gonna become very important come Assignment 4 to manipulate C strings, so that's why I want to do this.

So just imagine this right here being your main function. I don't care about Argc and Argv, but I do care about declaring one of these things – and I'll emphasize the fact that it is a string stack – and what I wanna do is I wanna press on four deep copies of these strings right here, const, char*. I'll just say letters is equal to – actually, you know what? Let me not call them letters. We'll just say friends – and I'll set it equal to this array. Now, Bob, Carl, and that'll be enough.

So what I wanna do is I wanna declare one of these stacks. This picture I get right here, according to that typedef over there, isn't very sophisticated. It's 16 bytes of nothing, or nothing meaningful, okay. But I rely on stack new ampersand of the string stack where I pass in size of char*, and all of a sudden now it's getting a little complicated.

I'm gonna ask the stack to basically keep track of the addresses of dynamically allocated C strings. So what I wanna do is I just got this picture. This points to a 16-byte blob. Four bytes – there's no – the depth – the stack is zero, but I have space for four elements. This four right here is really size of char*.

What I wanna do is I wanna for loop from i's equal to zero, i less than three because I wanna go ahead, and I wanna make a copy of each one of those strings. I do. That's char* – I'll call it Copy – is equal to strdup – oops – of friends of i, okay. This is an important enough variable that I actually wanna draw it, copy. On the very first iteration when i is equal to zero, it is set to point two, a deep copy of Al backslash zero in the heap.

What I wanna do – and I think this is the best way to phrase it – is that when you push an element onto the stack, you transfer ownership from you to the stack. The way you do that, based on this right here, is for me to do stack, push, which stack? The string stack that I've just initialized.

There's some drama and controversy over what the next argument should be. Since I am storing char*, that means that these things – even though the stack doesn't know it – that they have to hold as material these four byte character pointers. That means that I have to pass in the address of a char* so it knows to go to that address and copy the four bytes into the stack. Does that make sense to people?

Okay. Because I'm putting the ampersand here, it knows to go and replicate that material right there, so that it points there, increments this to a one. On the very next iteration, I reuse i, and actually destroy and re-declare Copy to no longer point to Al, but as it's re-declared and reinitialized it's set to point up to Bob.

This is Copy on the second iteration. I pass that in. It replicates the material that's in there because it has the address of that size of char* figure so that it could replicate that address right here, and so it's almost like this as a for loop blows up three balloons, okay, with Al, Bob, and Carl's name on it, and then knowingly memcpys the end of the string, or the tail of the string, and actually copies it to the stack behind the scenes. Do you understand what I mean when I say that? Okay. So, I'm really transferring ownership of these three dynamically allocated copies over to the stack.

What I wanna do now is I wanna go ahead and I wanna print these things out. I really wanna ask for ownership back, so I'm going to do this char*, name, and then a four int i is equal to zero, i less than three, i++.

What I wanna do is I wanna ask for the stack to pop off – string stack – and I want it to place the most recently pressed, or pushed, kite string back into the space that's right here, okay. So I want it to – this would have been set up to point to Carl. I want it to replicate this space in my local variable called Name so this ends up pointing to Carl, and the logical length of the entire stack is detrimental to two. Okay, does that make sense?

The way I do that is like this. Then I can do this. This is the equivalent in C of C out. [Inaudible] where this is the string percent. This is a placeholder for the string to be printed, and then I can go ahead and free main. That means that it basically passes the end of the kite string – or the end of the balloon string – to the deallocator, so it goes to the Carl address, or the Al address, or the Bob address, and actually donates that number back to the heap. I wanna be clean about it. I do stack dispose at the end, and I just pass an ampersand of string stack, and that is that, okay.

Now, these ampersands right here typically don't surprise people because I'm dealing with a direct allocation on the stack frame of this function of a thing called string stack, and I have to pass the location of it around, okay.

These ampersands right there very often surprise people. If you do not put them there, for many of the same reasons we saw in previous examples, it would still compile and run, but if you provide this address to stack push, then you're going to get it right. But if you actually don't include the ampersand right there, and you pass in that value right there, it's going to go ahead and replicate BOBY – I'm sorry – BOB backslash zero – as if it's an address, and copy that into the stack frame, okay. That's not what you want. Okay, does that make sense? Okay, very good.

Now, the problem comes – suppose I go ahead and I comment all of this out, or I set this equal to i less than two, or something. Suppose, at the time, I actually call stack dispose. The stack actually has some material that it still owns.

I've been very symmetric in the way that I allocate build up, bring down, and then dispose of, but the stack shouldn't be obligated to be empty, or the client shouldn't be forced to pop everything off the stack before they call dispose. Stack dispose should be able to say, "Okay, I seem to have retained ownership of some elements that were pressed onto me. I would like to be able to dispose of these things on behalf of the client before I go and clean up, and donate this blob of memory back to the heap." Okay.

Many times there's nothing to be done at all. When this thing stores ints, or longs, or doubles, or characters, you don't have to go in and zero them out. That's really not that useful. You do have to be careful about donating back any dynamically allocated resources, or maybe any open files. That's less common, but dynamic memory allocation is certainly more common.

If these things really are owned by the stack at the time it's disposed of, then the stack dispose function has to figure out how to actually pass these things right there to free just like we do right there. Does that make sense to people? Okay.

It's actually very difficult to do that because the implementation of stack dispose doesn't actually know that these things are pointers. It just knows, at best, that they're four-byte figures. It is capable of computing these addresses, and so if the depth of the stack is three, so that those three arrows are arrows that point to elements it's holding for the client. It could pass those three arrows to some disposal function, okay.

Now, this isn't always going to be a simple pointer. This might be a struct with three pointers inside, okay. It might be itself a pointer to a pointer to a struct that has three pointers inside. So, you wanna have some very general framework for being able to free whatever's at those three arrows if, in fact, there's anything freeing needs.

So, what I wanna do here is I want to upgrade this right here to not take two arguments, but to take three arguments, and this is what it's going to look like. This is the upgraded stack new function. Stack new is going to do that. It's gonna take elemsize, and it's also gonna take this, void. Free function takes a void* and doesn't return anything.

The idea here is that I want to pass to the constructor function information about how to destroy any elements that it holds for me when I call stack dispose, okay. Those three arrows – if you're dealing with a stack of depth three, and it's storing strings – it's prepared to pass those three things in sequence. At least, that's what I wanna write code for – those three things in sequence to this function that you write and pass the name of to your stack new function, okay. And it will invoke this function for every single element it holds for you.

Since we write this, we can accept the void*s right there. We interpret them, in this case, to be the char**s we know them to be, dereference them, and pass them to free, okay. Does that make sense to people? Yes? No? Okay.

So, I have to rewrite a few things. I have to actually store the free function as a field inside the struct. That means that this is a picture. We'll actually have this fifth field that points to a block of code that knows how to free things for me, okay.

We have to also handle the scenario where we're storing ints or doubles in the stack, and there actually is no freeing to be done on behalf of those things. So, the client, when they call this new function, they're supposed to pass on the first two arguments as they always have.

If you're storing these base types that have no freeing needs, I expect the client to pass an annul here, and that will be checked for and stack dispose. If you're storing things like char*s, or pointers to structs, or even direct structs that have pointers inside that are pointed to dynamically allocated memory, then you have a meaningful free function placed right here, okay. Does that make sense? Okay.

Let me rewrite stack new. I'm sorry, not stack new. This is easy. Let me rewrite stack dispose. Stack astro s understand that now I'm getting the pointer of one of these five field structures where the fifth field is actually either some null pointer, or a pointer to a legitimate freeing function.

Before I go ahead and dispose of the elems blob, I better check to see whether any – to see whether or not anything complicated is residing within the logLength elements that are still inside. If it is the case that s arrow free function – that's the name I gave to that field up there – I don't want to be too clever the way I do that, so let me say if it's not equal to null, then that means I have to apply this free function as a block of code against those three arrows, or all of the arrows that come up, the manual addresses – the manual the computer addresses of all the things that reside behind the scenes.

You could do this. Four int i is equal to zero, i less than s, logLength i++, I would just do s arrow, free function, char*, SRO elems plus i times s arrow elemsize. Okay. I think people don't usually argue with that because it's something they've seen before, so they kinda trust it. It's not difficult to get that part right when you know you're storing a free function. The part that's difficult to get right is right in the free function itself.

Let's revisit this example now that we know that when we store strings, we actually have to potentially set up the stack, or set up the stack to potentially delete elements for us. That means when we declare a stack called string stack, and we call stack new with size of char*, and I wanna pass in some function called string free – I'm just contriving the name. I do know that it has to match that right there as a prototype. It has to take a void* and return nothing.

I have the responsibility if actually writing the string free function. Well, I have to set up string free to actually accept the void* knowing that it's really a char**. Does that make sense to people? Okay.

Let's revisit this picture. These are the types of things that are gonna be passed to my free function. I need to reinterpret that as something that's at least dereferenceable, okay, and then hop into the actual rectangle, or the box, and take that number and pass it to the free function, okay.

So, as an aside, prior to doing this you would implement string three to take the void* – oops, let's give it a name – Elem, or VP, or whatever you wanna do – and because I'm writing this specifically for the char* stack case, I would just do this. This is really an asterisk. It just came out badly. Okay, that's a good one.

Okay, now what happens if I forget this, and I forget that? Think about what actually gets passed to free. If I don't cast this to at least be a double pointer – and I'm actually casting it to be a char double pointer because I know it's really two hops away as an arrow to characters – and dereference – this dereference is really what matters. It's the thing that takes me from this little fence post right there to the box that's addressed by the fence post, okay.

If I leave that that way, it will pass this address to free. It will pass that address to the free function. It will pass that address to the free function, and that's bad because the first one actually can be passed to free. You shouldn't be doing that, though, because you didn't allocate that block. These two addresses should certainly not be passed to free because they weren't directly handed back to anyone via up call to malloc or realloc, okay.

You don't own these copies of the pointers, but you know that they're char*s, and you're just telling the stack to actually dispose of those things for you because even though the stack isn't empty, you don't need the stack anymore. That's why it's imperative that these things really be there, okay.

If you're storing a stack of ints, or a stack of longs, or a stack of even struct fractions where there's no pointers inside, you would just pass in null there instead, and that would be special-cased away right there, okay. Does that make sense to people? Okay.

If I'm storing ints, or longs, or even struct fractions – which, when we double struct fractions just had two direct ints inside, if there's no dynamic memory allocation involved in the things I'm storing – even if they're structs – I would pass in null. The

constant right here is a sentinel meaning there's no free function needs, okay, and it's special-cased, and would be observed to be null right here, and circumvent this for loop, okay.

Just to make sure, let me ask some questions. Some of them were easy, but I wanna make sure you believe the answers. You understand why I for loop up to logLength and not allocLength, right? I have no business asking the stack to free things beyond the boundary between what's in use and what's not in use, okay. I have no reason to trust that anything meaningful is in that extra space. In fact, I know it isn't.

How come I don't free the element using this function right here just before stack pop exits? Do you understand what I mean? Like, if the stack is saying, "Oh, they want an element back. I better return this." Why isn't the stack applying the free function to the top element before it returns it? The way I framed it there it kinda sounds like an idiotic question. But you're not so much – you're not really transferring a copy of the string back. You're transferring ownership of the original string back to the client. You're taking this pointer and using memcpy to replicate it in client-supplied space.

So, if you do that and they have an alias to a pointer that you have an alias for, and then you apply the free function to it, you're killing the string one instruction after you actually give it back to the client, okay. Does that make sense? Okay, that's great.

Even if all the code makes sense, just be sensitive to the fact that the compiler does not help you out as much as we'd like. So, you really have to be very thoughtful about the placement of the ampersands, and the double asterisk versus the single asterisk, and whether or not you have to dereference a char** cast as opposed to not dereferencing a plain char* cast, okay.

The number of hops really matters, okay. And you always want to interpret the void*s to be the types of addresses you really know them to be, okay, because if you don't the composite is gonna let you do whatever you wanna do. And – well – I mean, in theory a compile time you can get away with a lot, but at run time you never get away with anything, okay. Does that make sense to people? Yeah?

Student: So, the free function will understand it's not just one character we're looking at; it's the whole string and the element?

Instructor (Jerry Cain):Well, that's not so much – that has nothing to do with string so much as it has to do with malloc and free, but the addresses that reside in this space right here, they were created using this function called strdup. I think the code's still up on the board right there, and strdup actually relies on malloc to allocate the memory.

Behind the scenes, even though it's unexposed to us, it actually keeps track of exactly how much memory is part of that blob. So, as long as you pass the leading address to it, it goes to a file where everything is kept track of, and it looks – it compares that address to

something in a symbol table, or something else to recover the actual allocation size so it knows exactly how much to donate back to the heap. Does that make sense? Okay.

Any other – the question way in the back.

Student: In the first line of int main, should that be a double char* [inaudible]?

Instructor (Jerry Cain):[Inaudible] it absolutely should be. This should be a double star. I meant to do this. Sorry. That's the way it is always in my sample code. I just forgot to do it here. Sorry about that. Was there another question that flew up over here? Yeah.

Student:[Inaudible]

Instructor (Jerry Cain):But I did, actually. This – there's a malloc that happens as part of that strdup. So, every single string has to be either freed by the stack itself as part of stack dispose, or if it comes back to me because I asked for via stack pop, and after I'm done with it I have to probably dispose of it. So there has to be a one-to-one correspondence between every call to strdup and every call to free.

In an ideal world where you only dispose of empty stacks, all the free calls would come in the client side. But if you ever dispose of a stack that still holds on to its elements – or is holding on to a subset of the elements – then the number of free calls is going to be distributed between the stack and the client. Does that make sense, okay. That's good. Okay.

So, what Assignment 3 is all about – it actually turns out that Assignment 3 used to be a very, very difficult assignment, but I started doing this in lecture about, like, I don't know, like three and a half years ago. And then all of a sudden, things became much more clear when it come assignment time because the implementation you write for the first half of Assignment 3 is really just an extension of this.

Rather than actually just dealing with push and pop as operations – those are the dynamic operations we're memcpy'ing those on – I want you to generalize it so that you can insert anywhere, and you're familiar with that from a data structure standpoint using the capital V vector from 106 or the lowercase v vector from the STL that you used in Assignment 1 and 2, okay. Does that make sense?

There are two things that I just wanna mention before I formally put this material to bed, and I wanna talk a little bit about the implementation of malloc, and free, and realloc and how they work. It's actually very interesting, I think. But I have to talk about two other functions that come up during the implementation of Assignment 3. I should just talk about them.

Let's – this is okay – I wanna write a function that's very similar to swap. I wanna write a function called rotate. This is actually imitates a function that's provided as part of the STL library, but I'm gonna write it in pure C, and I'm gonna pass in three void*s. Void*

– I'll call it front – void* – I'll call it middle – and void* – I'll call it end. And what I wanna do – I'll use this board to draw a picture – is I want front, middle, and end to actually be sorted pointers that point to various boundaries inside an entire array.

So let's assume I have an array of 50 integers, okay, and for whatever reason, I want to move the front four all the way to the back, okay. Does that make sense to people? So, think of it as, like, a bookshelf with 50 books on it, and for whatever reason the front four are out of alphabetical order, and you've decided to take them out as a chunk, and move them to the back, but in the process you have to slide 46 books forward. Now drop the word "book" and use the word "int" and that's what I wanna do.

The intent of this rotate function is if it's given the absolute opening address of the entire figure, it's given the midpoint that separates front from back – even though they're not equal sized – and then end is actually passed the end iterator in the C++ stance, but it's really the address of the first byte that has nothing to do with the array.

I can manually compute the number of bytes that's right there. I can manually compute the number of bytes that's right there from these three void*s. This is an implementation, needs to know nothing at all about the fact that that happens to be 50 ints over in that drawing. It could've been 200 characters, or it could've been 100 shorts, and it still should do the byte rotation in exactly the same way, okay.

The slight complication here that did not come up in the swap implementation is that this right here has to be written to temporary space. You're familiar with that idea from the swap implementation. Then this right here has to be memcpy'd – although that won't be the function we'll use. We'll see in a second why. This has to be memcpy'd from right here to right here. Does that make sense? Okay.

The problem with that is that – unlike all the other examples we've dealt with – the source region – this space, and this space right there – actually overlap, or can potentially overlap. Does that make sense to people?

The implementation of memcpy is brute force. It carries things four bytes at a time, and then at the end does whatever mod tricks it needs to copy off an odd number of bytes, but it assumes they're not overlapping, okay. When they're overlapping, that brute force approach might not work.

Suppose I wanna copy these first five characters – and this is meaningless, and this is meaningless. What memcpy would do – if I wanted to copy these five characters to right there – memcpy is actually quite careless – and it doesn't do exactly t his, but it does more or less this – where it would actually copy the A right there, and then copy the B right there, and then copy the C right there, and then copy the D right there, and copy the E right there, except that you've trounced on the C, D, and E before you had a chance to copy it. Does that make sense to people?

Now, memcpy could figure it out. It could actually check the start address – I'm sorry – the target address and the source address – and it could copy either from the front to the back or the back to the front, whichever direction is needed to ensure that it doesn't actually trounce over data before it's copied. Memcpy said, "I don't wanna bother with that. I wanna run as quickly as possible, and I want the client to take responsibility of only calling memcpy when, in fact, he or she knows that there's no overlap."

If they don't know if there's gonna be overlap they have to use a version of the function that's slightly less efficient, but does the error checking for them. Does that make sense? It has the same exact prototype – that shouldn't surprise you – but it's not called memcpy. It's called memmove.

I don't know why they use "move" as opposed to "copy." Probably because it's supposed to imply that it's actually shifting somewhere in – if you're dealing with overlapping ranges that you're really just moving bytes in a direction just by a little amount as opposed to really relocating them, okay.

So, the implementation of this has to be sensitive to that. What I wanna do is I wanna compute a few values. I wanna do int, front, size is equal to char*, middle minus char* front. Now, you look at that, and that looks a little weird. I am subtracting one void* from another. For the same reasons C does not allow pointer arithmetic – I'm sorry – pointer addition, it doesn't allow pointer subtraction either. Pointer subtraction you didn't deal with too much in 106B or 106X, but it is a defined, legal operation.

What it's supposed to do when you subtract one pointer from another – you may think that it returns the number of bytes that sits in between them. That's not true. If they're strongly typed to be int*s for instance – if you do pointer subtraction between two int*s it's supposed to return the number of ints that fit in between the two addresses, and that's consistent with the way that pointer addition works, okay. Does that make sense to people?

So, what I'm doing here is I'm – it's the same hack. I'm casting both of these things to be char*s so that pointer subtraction becomes regular subtraction, and I'm given the physical number of bytes that reside between that and that right there. Does that make sense? Okay.

I wanna do the same thing with end and middle so at least I know how big the two regions are. What I can do now is I can declare a raw buffer, just like I did with the generic swap function, char buffer, and I can allocate it to be of size front size. Again, this isn't ANSI standard, but it works on our compiler, and it's so much nicer that I'm just gonna do it.

So now I have a character buffer that's just as big as this thing is here, in terms of bytes. So maybe that's set aside right here because I said those were ints – if I draw it even close to scale – it will be of size 16 right there. And then I use memcpy.

I actually prefer to use memcpy, and I want you to use memcpy if you know you have the option to. I wanna memcpy into this buffer from front, front size so that if this as a figure happens to reside there, it's replicated right there. And if these are four ints, then this will eventually be able to stand in as four ints when it's placed in integer space.

Then I wanna take this and I wanna slide it down. When you call memcpy your heart's in the right place, but you're dealing with two overlapping figures so you wanna call – not memcpy – but memmove with the same signature, okay. That would be this – memmove. I wanna copy to front from middle, and I wanna copy back size, okay. Does that make sense to people?

Now, if mid is very close to the end, and it's beyond the 50 percent point, then it turns out that memmove didn't buy us very much because there's no overlapping figure – I'm sorry – there's no overlapping between the source region and the destination region.

But you can't – in a general sense – actually anticipate that, okay. You could actually look at how much closer – whether or not middle is closer to front or end – and then call one of two versions that only called memcpy, but then all you're doing is the error checking that memmove would do for you. So I'd rather just deal with one implementation, okay. Make sense? Okay.

The only complexity of the last line is getting this right here into the last front size bytes. I don't actually have a target pointer for this, but what I'll do is I'll memcpy – that's okay because I'm copying from the buffer – I'll do char*, end minus front size. If from here to here is front size, then from there to there is front size. I wanna copy from the buffer, and the size of the buffer is front size, okay. Does that make sense?

So, you only call memmove if you have to, okay, because you know that there's a very reasonable chance that the two – the source region and the destination region – will be overlapping. But I want you to call memcpy if you know you're able to because it's more efficient, and when you're starting to write systems code like this you actually do have to think a little bit about – more about – efficiency than you did in 106b.

People argue, "Well, why don't I just call memmove all time because then I can just forget." You're right. You could, but I'd rather you not. So I actually wanna pretend that memmove actually blows the computer up if the two regions don't overlap, okay, because I want you to call memcpy if you know you can, okay. Does that make sense to people? Okay.

The only other function I have to mention briefly – and I actually have plenty of time to do it so I'll just – but I just wanna go over the prototype of the generic quick sort function that exists in the C language, okay.

When you're sorting, there's not key. There's just the array, the element size, the number of elements, and then the comparison function is still relevant. You know how B sort only takes five arguments? Well, Q sort only takes four. It punts in the key. Everything

internally is a key compared to everything else, and it just uses the comparison function to guide it in doing all these generic swaps behind the scenes.

I do want you to use Q sort. I just wanna put the implementation up on the board so that I can say I've formally covered it, but you're all familiar with just sorting in general. Quick sort happens to be a very fast sorting algorithm. It has this as a prototype. Q sort takes a void* called base. It takes an int called n – or size, actually – an int called elemsize, and then it takes a comparison function that knows how to compare two generic addresses. And there's that. That's the prototype for it.

If you want more detail – this is even relevant in some degree to Assignment 2 – if you happen to be stuck on using B sorts and you just wanna use some more details, you can – at the command prompt – type in MAN Q sort, and you will get more information about Q sort than you care to get. But nonetheless – at the top – will remind you what the prototype is, and what all the arguments are named so you know which ints correspond to which.

You can do MAN on B search – MAN is short for manual so it just basically gives you a textual documentation of that function. Also, memcpy, memmove – and I think there's some other ones – malloc, realloc, free. These are the types of functions that are exercised aggressively by Assignment 3 so you'll definitely want some resource to go to if it's 5 a.m., and you're working on it, and there's nobody around, okay. Does that sit well with everybody? Okay.

So you have plenty of time to do Assignment 3. It has turned out to be – it is – I won't say it's easy by any stretch of the imagination because I don't have – there's no advantage to me saying that, but it is actually a little bit less work than Assignment 2. And people always start on it with a lot of enthusiasm because they think there's a lot of work to be done – and there actually is – but there's a clear list – a to do list of things – there's like 13 functions that have to be implemented, and I provide all kinds of unit tests to exercise all of these things, okay.

And you will just make very piecemeal progress on a nightly basis, okay, so that you can get it done in one or two nights if you actually know what's going on, okay. So just give yourself a little bit of a buffer in case you have some gotchas that come up while you're implementing. But it is – usually surprises people that it's not as much coding as Assignment 2 is, okay.

What I wanna do now is I have ten minutes. I just wanna give you a little preamble to the more – the more involved lecture I'm gonna give on Friday about the implementation of malloc, and realloc, and free. Let me draw what – for the first time of many –is gonna be my generic drawing of all the memory.

Here's RAM, and since we're dealing with a – since we're dealing with an architecture where longs and pointers are four bytes, that means that pointers can distinguish between two to the thirty-second different addresses. That means that the lowest address in

memory is zero – which is that null that you're starting to fear a little bit – and then the highest address is two to the thirty-second minus one, okay.

Whenever you call functions, and the function call forces the allocation of lots of local variables, those local variable – or I'm sorry – the memory for those local variables is drawn from a subset of all RAM called the stack. I'm gonna draw that up here. I drew it a little bit bigger than I needed to, but here it is. The stack segment is what this thing is called.

It doesn't necessarily use all of the stack, but for reasons that will become clear – and there's even a little bit of intuition, I think, as to why it might be called a stack – when you call main you get all of its local variables, and they're alive and they're active, okay.

When main calls a helper function it's not like main's functions – main's variables – go away. They're just temporarily disabled, and you don't have access to them – at least not via the normal variable names, right. So main calls helper, which calls helper helper, which calls helper helper helper, and you have all of these variables that are active – I'm sorry – that are allocated, but only the ones on the bottom most function are actually alive and accessible via their variable names.

When helper helper helper returns, you return back to the local where helper helper has local variables, and you can access those, okay. So basically, when a function calls another function, the first function's variables are suspended until whatever happens in response to the function call actually ends, okay. And it may itself call several helper functions, and just go through lots of – a big code tree of function calls before it actually returns a value, or not, okay.

What happens is that, initially, that much space is set aside from the stack segment to just hold the main's variables – whatever main's local variables are. And when main calls something, this threshold is lowered to there to just make sure that not only is there space for main's variables set aside, but also for the helper function's variables, okay.

And it goes down and up, down and up, every time it goes down it's because some function was called, and every time it goes up it's because some function returned, okay. And the same argument can be made for methods in C++.

It's called a stack because the most recently called function is the one that is invited to return before any other ones unless it calls some other function, okay. That's why it's called a stack.

We'll get to this – we'll probably spend two or three lectures talking about not only how this thing's formatted, and how variables are laid out, but also how assembly code actually manipulates the information in here. And assembly code even actually decrements and increments this boundary for us in response to function call and return.

What I wanna focus on is this other block of memory called the heap segment. The fact that the heap and the stack are date structures from cs106b is actually irrelevant here – mostly irrelevant, okay.

Heap in this world doesn't mean like a priority cube back in data structure. It really means blob of arbitrary bytes that this is the lowest address, this is the highest address of the heap segment as opposed to this segment right here, which is completely managed by the hardware – by the assembly code, which actually happens to be down here, okay – this right there, this boundary, and that address, and that address is admitted to software – software that implements what we call the heap manager, okay.

And the heap manager is software – it's code. The implementation for malloc, realloc, and free are all written in the same file, and they basically manage this memory right here, okay.

So, every time you call malloc of 40, it goes, and virtually finds a block of size 40 in this, and seemingly returns the lead address of it, okay. If you haven't freed 40 yet, and you go and allocate space – or malloc a request for 80, it might draw it from here – it's a little bit more organized than this. It doesn't just draw it from a random location, but malloc would return that.

If you realloc this pointer right here, and you ask for it not to be 40 anymore, but to be 100 because of the way it's drawn it actually might just extend this to be a blank 100, and not touch the bytes that are up top – up front. If you reallocate this, and you ask for 8,000, and it doesn't happen to have 8,000 minus 80 bytes between that boundary and that boundary, it might go and allocate a blob that's 8,000 bytes, copy this over, and copy it right – and then free this right here.

So this really is a sandbox of bytes, and whether they're integer arrays, or char* arrays, or struct fraction arrays, it's all immaterial to malloc. It only allocates things in terms of numBytes requests, okay. Does that sit well with everybody? Okay.

So what I wanna do is I wanna be a little bit more organized in the way I demonstrate how malloc, and realloc, and free work because it isn't that rain pell mell about allocating bytes. It does have some normative process that's followed behind the scenes so it can be as efficient as possible on your behalf. Malloc, and free, and realloc are actually called enough – either directly or via things like operator new, and operator delete, or strdup, or your constructors, or whatever, but it wants to make them run as quickly as possible.

So what I'm gonna do is I'm gonna explode this picture to this board over here – actually, this one's better – but I'm gonna emphasize the fact that it is one big linear array of bytes. And so, rather than drawing it as a tall rectangle, I'm gonna draw it as a very wide rectangle, and make it clear that this address right there is that one right there, and this address right there is that right there, okay. Does that make sense to people?

So I go ahead and I declare void* A is equal to malloc of 40. This isn't exactly what happens, but it's pretty close. It will usually search from the beginning of the heap, and look for the smallest – I'm sorry – the first free block of memory that's able to accommodate this size request.

And initially, since the entire heap is available – what'll happen is it'll do whatever accounting behind the scenes as is necessary to clip off the first 40 bytes, record that it's in use somehow – we'll talk about that in more detail on Friday – and return the address of that right there. Does that make sense? Okay.

Next line is this. Malloc of 60 able – take this least – this naïve approach, and that's what it does very often. It just scans from the beginning of the heap, and find the first open block that's able to meet this request. It sees that that's in use. It's able to quickly hop here – we'll see why on Friday – and say, "Okay, this entire thing is free. That's certainly bigger than 60. So I will do that, and then return that address." Okay.

Punting on realloc for a second, if I go ahead and call free on A it will go in because this address – this arrow right there – the tail of it is held in the A variable.

It will go and remove the halo around this memory – it doesn't clear out the bit patterns because the bit patterns are supposed to stop mattering – and then donates it back. So if I do this, void* C is equal to malloc, and I'll [inaudible] notches about it, and I say 44, it will look at this block right here. It will record it, and note that it's a 40 byte free block that could've been used had this number been less than or equal to 40, but since it isn't it has to hop over and consider this right there. So it will clip this off and return that address and bind it to C.

If on the very next line I do this – some implementations actually carry off where the last search ended, but the way I'm talking about it, it always searches from the beginning, okay. This time this block is big enough to meet that size request so it might clip this off, record that it's 20 bytes wide, and return that pointer again, okay. Does that make sense?

Entirely software managed with very little exception, okay – and I say exception because the operating system and what's called the loader has to admit to the implementation what the boundaries of the stacks of the heap segment are – but everything else is really frame in terms of this raw memory allocator, okay.

As far as realloc is concerned, if I pass this address to realloc, and I ask it to become bigger, it'll have to do a reallocation request, and put it somewhere else – probably right there, the way we've been talking about it. If I didn't ask for that to be realloced, but I asked for this to be realloced – to go to 88 – it would just extend it in place.

What's gonna happen – and we'll be more detailed about this come Friday – is that there really is a little bit of a data structure that overlays the entire heap segment, okay. It is manually managed using lots of void* business.

In a nutshell – let me actually, in the final ten seconds here – let's say that this is the heap again – and just to emphasize what's been allocated and what hasn't been, let's say that this has been set aside. Let's say this has been set aside, and let's say that this has been set aside. The data structure that's more or less used by the heap manager overlays a linked list of what are called free notes, okay.

And it always keeps the address of the very first free note, and because you're not using this as a client, the heap manager uses it as a variably sized node that – right in the first eight or four bytes – keeps track of how big that node is. Does that make sense to people?

So, it might have subdivided that, and to the left of that line might have the size of that node, and to the right of that line might actually have a pointer to that right there, okay. Now it's not like there's ints and doubles typing any of the material over here. The heap manager has to do this very generically so it constantly is casting addresses to be void*s and void**s, okay, in order to actually jump through what this thing called the free list behind the scenes to figure out which node best accommodates the next malloc request, okay. When you free this node right here it has to be threaded back into the free list, okay. Does that make sense?

I'll be a little bit more detailed come Friday as opposed to this hand wavy after 11:50 comment, okay. But I want you to understand all this. Okay, see you on Friday.

[End of Audio]

Duration: 53 minutes