ProgrammingParadigms-Lecture06

Instructor (Jerry Cain):Hey, everyone. Welcome. I have one very short handout for you today. It is the handout that has two problems that we'll be going over during tomorrow afternoon's discussion section. Remember, we don't have it at 3:15; that was just last week to accommodate what I assumed would be a large audience. It's at 4:15. It's actually permanently in the room down the hall, in Gates B03. So I'm not teaching the section, Samaya is, who I think is here, who is handing out the handout. So look for that guy tomorrow at 4:15.

Both of these are all the exam problems, so they're certainly good problems to understand the answers to. And if you're not gonna watch the section or attend, just make sure you at least read the handouts. You may very well be able to do the problems yourself, and if so then that's fine, but if not, make sure you watch the discussion section at some point.

When I left you on Friday, I was a quarter way, a third a way through the implementation of an int-specific stack. So there's nothing technically cs107 level about this, except for the fact that we're being careful to write it in pure C, as opposed to C++.

Now, if you remember the details from Friday, about the interface file, the functions are, I mean, I'll talk about those in a second. We actually expose the full struct. There are no classes, and there's no public, and no private. So everything is implicitly public in the dot age. That's a little weird. C++, when you learned about classes and objects, it was all about encapsulation and privacy of whatever could be private. We can't technically do that in C; although, we can certainly write tons and tons of documentation saying just pay attention to the type, use these functions to manipulate it, and pretend that these are invisible to you.

You more or less operate as if this is a constructor, a destructor, and methods, but they happen to come in pure, top-level function form, where you pass in the structure being manipulated. So StackNew is supposed to take the struct, one of these things, addressed by s, and take it from a 12-byte block of question marks to be logically representing a stack of depth zero. This is supposed to kill a stack at that address. This is supposed to increase its depth. This is supposed to pop something off.

I wrote StackNew and StackDispose. I certainly wrote StackNew, I'm forgetting about StackDispose, but I'll write them really quickly right now.

This is the dot c file:

Stack .c

void StackNew

Stack

I add these three things:

Stack logLength = 0 (that's because the stack is empty)

I'm going to make space for four elements, and then I'm going to go ahead and allocate space for four elements using c as a raw dynamic memory allocator, called malloc. It doesn't take anything related to a data type. It doesn't know that anything related to a data type is coming in as an argument. You have to pass in the wrong number of bytes that are needed for your four integers. And that's how you do this. All that malloc feels is an incoming 16. It goes to the heap and finds a figure that's that big, puts a little halo around it saying it's in use, and then returns the base address of it.

I told you to get in the habit of using a certain macro, that was mentioned a little bit in Assignment 1, just to confirm that elems ? null. If malloc fails for some reason it rarely will fail because it runs out of memory, it's more likely to fail because you called free on something you shouldn't have called freed on. So you've messed up the whole memory allocator behind the scenes.

That's a good thing to do right there because it very clearly tells you where there's a problem, as opposed to it just seg filtering or bus erroring or it crashing in some anonymous way, and you have not idea how to back trace to figure out where the problem started.

As far as stackDispose is concerned, this is trivial. Although, there's one thing I want to say about it, actually, two things I can think of. I want to go ahead and do the opposite of malloc. This corresponds to operator delete from C++. I want to free s-> elems so that it knows that whatever figure is identified by that address right there should be donated back to the heap. That's just what I mean when I talk about free right here. Some people question whether or not I should go and actually free s itself. They ask whether or not that should happen. The answer is no.

Nowhere during StackNew did you actually allocate space for a stack. You assumed that space for the stack had been set aside already, and that the address of it had been identified to the StackNew function. So you don't even know that it was dynamically allocated. In fact, the sample code I wrote last time declared a stack called s as a local variable. So it isn't dynamically allocated at all. So you definitely in this case do not want to do this.

The other thing I want to mention is that regardless of whether or not the stack, at the moment this thing is called, is of depth zero or of depth 4500, the actual ints that are held inside the dynamically allocated rectangle of bytes, there's no reason to zero them out. And there's certainly no freeing needs held by those integers. I say that because just imagine this not being an int stack but imagine it being a char * stack, where I'm storing dynamically allocated c strings, Freds' and Wilmas' and things like that.

If I did have that, in the case of the char *, I would have to for loop over all the strings that are still held by the stack at the time it's being disposed and make sure I properly dispose of all of those strings. I don't have any of that with the int specific version. But the reason I'm saying this is it's going to have to accommodate that very scenario when we go generic on this thing and just deal with blobs as opposed to integers.

The most interesting of the four functions is the StackPush. And it's interesting not because of the algorithm to put an integer at the end of an array, but the algorithm that's in place to manage the extension of what's been allocated to be that much bigger because you've saturated what's already there.

I chose an initial allocated length of four, so I'm certainly able to push four integers onto the stack and not meet any resistance whatsoever. But if I try to press a fifth in on the stack, it has to react and say, I don't have space for five integers I better go and allocate space for some more, copy over whatever's been pushed, and dispose of the old array to make it look like I had space for 8 or 16 or 1,024 or whatever.

So the implementation, assuming I do have enough space, would be this simple: StackPush. Pushing onto what stack? The one that's addressed by this variable called s. What number am I pushing on? The one that comes in via the value parameter.

Let me leave some space here for what I'll inline as the reallocation part. But if there's enough memory, where down here I can assume that there's definitely enough memory, I can do this:

s-> elems of s-> logLength equals this value

Think about the scenario where the stack is empty. The logLength happens to be zero, which happens to be the index where you should insert the next element.

Once I do this for the next time, I have to go ahead and say, you know what, the logLength just increased by one. Not only does that tell me how deep the stack is, it also tells me the insertion index for the very next push call. It isn't that simple. Eighty percent of the code that gets written here has to deal with the one in two the end time scenarios where you actually are out of space.

If it is the case that s-> logLength == s-> allocLength, then as an implementation you're unhappy because you're dealing with a stack and a value where the value has no home in the stack at the moment.

I'm trying to press on a 7. Let's say that s points to this right here, and the logical length and the allocated length are both 4, and I've pushed these four numbers on there, the client doesn't have to know that there's a temporary emergency. So right here I'm just gonna react by saying, you know what, that 4 wasn't big enough, I want to go ahead and I want to, without writing code yet, I want to basically reallocate this array. Now, I say reallocate because there really is a function related to that word in the standard C library.

In C++ you have to go ahead and allocate an array that's bigger. You don't have to use a doubling strategy; I'm just using that as a heuristic to allocate something that's twice as big. You have to manually copy everything over, and then you have to dispose of this after setting elems equal to the new figure.

It turns out C is more in touch with exposed memory than C++ is. So rather than calling malloc yourself, you can call a function that's like malloc except it takes a value, that's been previously handed back to you by malloc, and says please resize this previously issued dynamically allocated memory block. That is this:

Let me write just one line right here. I want to take allocLength and I want to double it. I could do plus equals 10; I could do plus equals 4. I happen to use a doubling strategy. And then what I want to do is I want to call this function. Let me deallocate these so I have space to write code.

s-> elems equals this function called realloc

There's no equivalent of this in C++. I'll explain it in a few weeks why there isn't. But realloc actually tries to take the pointer that's passed in and it deals with a couple of scenarios. It sees whether or not the dynamically allocated figure can be resized in place, because the memory that comes after it in the heap isn't in use. There's no reason to lift up a block of memory and replicate it somewhere else, to resize it if the part after the currently allocated block can just be extended very easily in constant time. The second argument to realloc is a raw number of bytes. So it would be:

s-> allocLength times size of integer.

I see a lot of people forgetting to pass in or forgetting to scale this byte size event. Even though that they know to do it with malloc, they forget with realloc for some reason. It takes this parameter right here, it assumes it's pointing to a dynamically allocated block, and it just takes care of all the details to make that block this big. If it can resize it in place, all it does is it records that the block has been extended to include more space and it returns the same exact address.

So that deals with the scenario where this is what you have, this is what you want, and it turns out that this is not in use so it can just do it. So it took this address in and it returns the same exact address.

Well, you may question why does it return an address at all? In this case it doesn't need to. However, it may be the case that you pass that in and you want to double the size or make it bigger, and that space is in use. So it actually does a lot of work for you there. It says, okay, well, I can't use this so I have to just go, and it really calls malloc somewhere else on your behalf. Just assume that's twice as big as this. Whatever byte pattern happens to reside here is replicated right there. The remaining part of the figure isn't initialized to anything, because it's uninitialized just like most C variables are. It actually frees this for you and it returns this address.

So under the covers, beneath the hood of this realloc function, it just figures out how to take this array and logically resize it, and preserves all the meaningful content that's in there. If it has to move it it does free this. It turns out realloc actually defaults to a malloc call if you pass in null here. So technically you don't need the malloc call ever. That actually turns out to be convenient when you have entered a processes that have to keep resizing a node, and you don't want a special case it and call it malloc the very first time, you can just call it realloc every single time when the first parameter is null on the very first iteration.

It's very easy to forget to catch the return value. If you forget to do that then you refer to s-> elems captures the original address, which may have changed, and you now after this call may be referring to dead memory that's been donated back to the heap manager. So it's very important to catch it like this. If realloc fails it'll return null. So I'm gonna put an assert right here.

The one thing about realloc that's neat is that if you don't want to just end the program because it failed, it actually, if it returns null it won't free the original block, it would only return null if it actually had to move it. Actually, that's not true. If it can't meet the realloc page in request it'll just leave the old memory alone and return null. In theory, you don't have to assert. And in the program here you could just check for null, and say, okay, well, maybe I won't resize it, maybe I'll just print a nice little error message saying, I actually cannot extend the stack at this time, my apologies, please do something else.

We'll learn a lot about how malloc and realloc and free all work together. They're implemented in the same file. They're implemented in the same file because even though that the actual blob of memory doesn't expose its size to you, like in a dot length field like in Java, somehow free knows how big it is. Well, there's cataloging going on behind the scene so it knows how much memory to donate back to the heap every time you call free. But I don't want to focus on that.

This right here, it doesn't get called very often. This doubling strategy is popular because it only gets invoked as a code block one out of every two to the n calls once you get beyond 4. Because of the doubling strategy [inaudible] only comes up once every power of 2; 512 wasn't big enough, okay, well, maybe 1,204 will be. And you have a lot of time before you need a second reallocation request.

There are a couple of subtleties as to how this is copied. In this example right here, all of these little chicken scratch figures right there, they correspond to byte patterns for integers. So when I replicate them right there, I trust that the interpretation of those integers right there will be the same right there. So realloc is really moving my integers for me.

If these happen to be not four integers but four char *'s, that means the byte patterns that were here would have actually been interpreted as addresses. When it replicates this byte pattern down here, it turns and it replicates the addresses verbatim. This would point to

that; that will point to that; this will point to that; this will point to that. Do you understand what I mean when I do that?

When I dispose of this it doesn't go in and free the pointers here. It doesn't even know there are pointers in the first place so how could it free them? So as this goes away, and these all point to where other pointers used to be pointing, you don't lose access to your character strings.

This is certainly the most involved of all four functions.

Student: If you could explain the assertion again.

Instructor (Jerry Cain):All assert is, for the moment just pretend that it's a function. It takes a boolean. Its implementation is to do absolutely nothing and return immediately if it gets true, and if it gets false its implementation is to call exit after it prints an error message saying an assert failed online, such and such, in file stack dot c.

Student: So actually [inaudible].

Instructor (Jerry Cain):S-> elems ? null. You want to assert the truth of some condition that needs to be met in order for you to move forward. If realloc fails it will return null. You want to make sure that did not happen. That's why there is a not equals there as opposed to a double equals.

This isn't technically a function. We'll rely on that knowledge later on. It's technically what's called a macro. It's like a #define that takes arguments. But, nonetheless, just pretend for the moment that it works like a function.

Student: If the realloc has to move the array is it now or anytime?

Instructor (Jerry Cain):Yeah. The question is if realloc has to actually move the array it is time consuming. It's even more time consuming than o of m. It actually involves not only the size of the figure being copied, but the amount of time it takes to search the heap for a figure that big. So it really can, in theory, be o of m, where m is the size of a heap.

Let me just quickly, just for completeness, write stackPop. It turns out it's not very difficult. The most complicated part about it is making sure that the stack isn't empty.

This continues on this board right here. I have this thing that returns an int stackPop. I'm popping off of this stack right here. No other arguments assert that s-> logLength is greater than 0. If you get here then you're happy because it has something you can pop off. So go ahead and do the following:

s-> logLength -- and then return s-> elems of s-> logLength

The delta by one and the array access are in the opposite order here. I think for pretty obvious reasons. You're effectively control z'ing or command z'ing the stack to pop off what was most recently pushed on. So if this came most recently, you want to say I didn't even do that. Oh, and by the way, there's the element that I put there by mistake. That's what I mean.

You could say you see this as demanding a reallocation request and going from 100 percent to a new 100 percent, where the old 100 percent was actually 50 percent. You may ask whether or not if you fall below a 50 percent threshold that you should reallocate and say, oh, I'm being a memory hog for no good reason and donate it back to the heap. You could if you want to.

My understanding of the implementation of realloc, because it wants to execute as quickly as possible, it ignores any request to shrink an array. As long as it meets the size request, it doesn't actually care if it allocates a little bit more. So it's like, oh, the size is 100 bytes and you just want 50. Okay, then only use 50 but I'm gonna keep 100 here because it's faster to do that.

With regard to the stackPop prototype right there, it sounds like a dumb observation, but it'll become clear in a second when we go generic. This particular stackPop elects to return the value being popped off. I can do that very easily here because I know that the return value is a four-byte figure. If this were a double stack I would just make that leftmost int up there double and it would just return an eight-byte figure.

When I go generic, and I stop dealing with int *'s and I start dealing with void *'s, I'm gonna have to make the return type either void *, which means I return a dynamically allocated copy of the element, for reasons that'll become clear in a little bit I'm not gonna like that, or I have to return void and return the value by reference by passing in an int * or a void * right here. That will become more clear once I actually write the code for it. But we take advantage of a lot of the fact that we know that we're returning a four-byte figure here so the return type can be expressed quite explicitly as in int.

So now what I want to do is I want to start over. And I want to implement all of this stuff very generically. And I want to recognize that we're trying to handle ints, and bools, and doubles, and struct fractions, and actually the most complicated part of it are char *'s because they have dynamically allocated memory associated with them.

Let me redraw stack dot h. And we are going completely generic right here. Most of the boilerplate is the same. Typedef struct, and I don't want to commit to a data type, elems. Now, think about what I lost. I lost my ability to do pointer arithmetic without some char * casting. I also lost intimate knowledge about how big the elements themselves are. So I can't assume the size of int anymore because it may not be that; it probably won't be. So I'm gonna require the prototype of StackNew to change, to not only pass in the address of the stack being initialized, but please tell me how big the elements that I'm storing are so that I can store it inside the struct.

The logical length versus the allocated length and the need to store that, that doesn't change. I still want to maintain information about how many of these mystery elements I have. I also want to keep track of how much I am capable of storing, given my current allocation. And there's gonna be one more element that we store on a byte. So I'll leave that piece of spencil for the next 15 minutes and we'll come back to it.

The prototype of the functions change a little bit. StackNew, same first argument, but now I take an elemSize.

Void * stackDispose

stack *s (That doesn't need to change. It's mostly the same, although, we'll have something to say about what happens when it's storing char *'s.)

Void stackPush

stack s

And then I'm gonna pass in void * called elemAddr.

I can't commit to a pointer type that's anymore specific right there 'cause it may not be an int *, it may not be a char **, it may not be a structFraction *. It just needs to be some address that I trust because the information that I am holding is pointing to an s-> elemSize figure. So there's that.

This is the part that freaks people out. This is what I'm going to elect to do, I'll talk about the alternative in a second. But when I pop an element off the stack, I want to identify which stack I'm popping off of, and I also want to supply Address. This is me supplying an address. I'm actually gonna identify a place where one of my client elements, that was previously pushed on, should be laid down. It's like I'm descending a little basket from a helicopter so somebody can lay an integer or a double or something in it so I can reel it back up to me. So void * elemAddr. And that's all I'm gonna concern myself with.

Student: Where is the struct named stack?

Instructor (Jerry Cain):I'm sorry, I just forgot it. It's right there.

So 70 percent of the code I'm gonna write it's all gonna be the same. It's gonna be slightly twisted to deal with generics as opposed to ints. But rather than using assignment, which works perfectly well when you're taking one space for an int and assigning it to another int, we're gonna have to rely on memcopy and things like that.

StackNew is not hard.

Void StackNew

stack *s int elemSize (Same kind of stuff)

s-> logLength = zero

s-> allocLength = four

s-> elemSize = elemSize

s-> elem = malloc

I don't have a hard data type so I can't use size of here, but I have no reason to. I have been told how big the elements are via the second parameter. So four times elemSize, just use the parameter as opposed to the field inside the struct.

I can do a few things to make my life easier. S-> elems better not be equal to null or else I don't want to continue. And the assert will make sure of that. I also could benefit by doing this: assert that s-> elemSize – this one is not as important because it takes a lot of work to pass in a negative value for elemSize. But, nonetheless, it's not a bad thing to put there because if you try to allocate -40 bytes it's not gonna work. That and implementation make sense.

I think even if it makes sense, there's some value in seeing a picture. It takes this stack right there, with its four fields inside at the moment, and let's say I want to go ahead and allocate a stack to store doubles. That means the elemSize field would have an eight right there. I'd set aside space for four of them. The logical length is zero. I'm just making sure this is consistent with the way I've done this. That's right. And then I would set this to Point 2. As far as the stack knows, it has no idea doubles are involved, it just knows that it's a 32-byte wide figure. And it has all of the information it needs to compute the boundary between zero and one, one and two, and two and three.

As far as stackDispose is concerned, stack *s, this is incomplete, but it's complete for the moment given what I've talked about. I just want to go ahead and I want to free s-> elems, and not concern myself yet with the fact that the actual material stored inside the blob might be really complicated. Just for the moment think ints, and doubles, and plain characters. But the fact that I'm leaving space there [inaudible] that this will change soon.

Let me write stackPush right here.

void stackPush

stack *s

void * elemAddr

I'm gonna simplify the implementation here a little bit by writing a helper function. I could have written the helper function on the int specific version but just didn't.

Up front, if it's the case that s-> logLength == s-> allocLength, then I want to cope by calling this function called stackGrow. And you know what stackGrow's intentions are. And I'm just gonna assume that stackGrow takes care of the reallocation part. I'll write an int second, that's not the hard part. Once I get this far, whether or not stackGrow was involved or not, I want to somehow take the s-> elemSize bytes that are sitting right there and write into the next slot in memory that I know is there for me, because if it wasn't this would have been called.

So what has to happen? Let me refer to this picture. Let's just assume that this picture is good enough, because I obviously have enough space to accommodate this new element. Let's say that I have three elements. So that these have been filled up with interesting material. And I somehow, in the context of that code over there, have a pointer to some other eight-byte figure that needs to be replicated in.

This is gonna be the second argument to memcopy. This right there is gonna be the third argument. The only complexity is computing and figuring out what the first argument is. It has to be that. It doesn't need to know that it's a double *, or a struct with two ints inside *, or whatever. It just needs to actually get the raw address and replicate a byte pattern. No matter what the byte pattern is, as long as it's the same in both places, you've replicated the value.

So the hardest part about it is doing this:

void * target = char * s-> elems + s->

logLength times s->elemSize

This is why I demanded all of this stuff to be passed to my constructor function so it was available to me to do the manual pointer arithmetic later on.

I don't think I messed this up. LogLength is right there for the same reasons it was in between the square brackets for the int specific implementation of this.

Then what I want to do is I want to go ahead memcopy into the target space whatever's that elemAddr. How many bytes? This many, s-> elemSize. Then I can't forget this, this was present at the other implementation as well. I have to note the fact that the logical length just increased by one. So that's how I managed that.

Do you guys see just the bytes moving on your behalf in response to this function?

There's a question back there.

Student: The char * after void target equals?

Instructor (Jerry Cain):This right here, remember that s-> elems, unless I made a mistake, is typed to be a void *. So you can't do pointer arithmetic on the void *, so the

trick – there's actually two tricks. I opt with this one because I'm just more familiar with it, you can either cast it to be a char * so that pointer arithmetic defaults to regular arithmetic. This as an offset it's still the offset, it just happened to involve multiplication. It is itself implicitly multiplied by size of char, which as far as multiplication is concerned, is a no op.

I have also seen people, I think I mentioned this before, I've seen people cast that to be an unsigned long, so that it really is just plain math, and then they cast it to be void *. Pointers and unsigned longs are supposed to be the same size on 32-byte systems. A long is supposed to be the size of a word on the register set. So I've seen that as well. You can do whatever you want, I just – all of my examples use the char * version so that's why I use that one.

The stackGrow thing. The reason I want to do that is not because of the mem copying or the void *, I just want to explain what a student asked two seconds before class started. You're kind of already getting the idea that the word static has like 85 meanings in C and C++. Well, here's one more.

When you see the word static, decorating the prototype of a C or a C++ function, not a method in a class just a regular function, such as static void stackGrow, and it takes a stack *s. What that means is that it is considered to be a private function that should not be advertised outside this file. So in many ways it means private in the C++ sense. The technical explanation is that static marks this function for what's called internal linkage.

You know how you're generating all these dot o files, when you type make all this stuff appears in your directory, some of them are dot o files. I'll show you a tool later on where you can actually look at what's exported and used internally by those dot o files.

StackPush, and stackNew, and stackDispose are all marked as global functions, and that the symbols, or the names of those functions, should be exported and accessible from other dot o files, or made available to other dot o files. Something like this is marked as what's called local or internal. And even though the function name exists, it can't be called from other files. That may seem like it was a silly waste of time, but it really is not.

Because you can imagine in a code base of say one million files, it's not outlandish believe it or not. Think Microsoft Office, the whole thing, probably has on the order of hundreds of thousands of files, maybe tens of thousands, I don't know what it is, more than a few. You can imagine a lot of people defining their own little swap functions. And if they're not marked as internal functions at the time that everything is linked together to build Word or Excel or whatever, the linker is gonna freak out and say, which version of swap do I call? I can't tell. Because it has 70 million of them. But if all 70 million are marked as private, or static, then there's none of those collisions going on at the time the application's built.

This is responsible for doing that reallocation. Assume it's only being called if it understands that this is being met as a precondition. So it can internally just do this:

s-> elems = realloc

s-> allocLength times s-> elemSize

That's the cleaner way to write it. It makes this focus on the interesting part that's hard to get, and kind of puts this aside as uninteresting.

Algorithmically, the function that's most different from the integer version actually is this StackPop call. This is how I want to implement it:

void stackPop.

I'm popping from this stack right here. I'm placing the element that's leaving the stack and coming back to me at this address. There's no stack shrink function to worry about. So what I want to do is declare this void *, not called target but called source = char * of s-> elems plus s-> logLength minus 1 times s-> elemSize. I forgot to do the minus minus beforehand so I recovered by doing a minus 1 right there.

This is where we're drawing a byte pattern from in the big blob behind the scenes. We're drawing byte patterns from there so we can replicate it into elemAddr. ElemAddr is the first argument. I'm copying from that address right there how many bytes. This many. And then do what I should have done earlier, logLength --. This really should have been the first line, and I shouldn't have the -1 there, but this is still correct.

Do you guys get what's going on here? If you understand the helicopter basket analogy? We're on actually identifying a space where it's safe to write exactly one element so that when the function returns I can go, wow, that's been filled up with an interesting element to me. And I can go ahead and print it out or add it to something or replace the seventh character or whatever I want to do with it.

This used to be int. If I wanted to I could have punted on this right here and just passed in one argument. And I could have returned a void * that pointed to a dynamically allocated element that's elemSize bytes wide. And I just would have copied not into elemAddr, but into the result of the malloc call. With very few exceptions, malloc and strdup and realloc being them, you usually don't like a function to dynamically allocate space for the call and then make it the responsibility of the person who called the function to free it.

There's this asymmetry of responsibility, and you try to get in the habit as much as possible of making any function that allocates memory be the thing that deallocate's it as well. There's just some symmetry there and it's just easier to maintain dynamically allocated memory responsibilities.

It's not wrong it's just more difficult to maintain. It actually clogs up the heap with lots and lots of little void *'s pointing to s-> elemSize bytes, as opposed to just dealing with the one central figure that's held by the stack, and then locally defined variables of type int and double that are passed in as int * and double *'s recognized as void *'s. But

because we laid down the right number of bytes, as long as everything is consistent we get back ints and doubles and things like that.

Student: In this case do you have to with our elemAddr [inaudible] right?

Instructor (Jerry Cain):Yeah. I'm not actually changing elemAddr I'm just changing what's elemAddr. So let me actually make a sample call to this.

Suppose I have a stack s. And I called StackNew with & of s, and I pass in size of int. That means that I have this thing, that's my stack and I'm supposed to just take it as a black box and not really deal with it. But I know behind the scenes that it has a 4 there, and maybe my stack has 14 elements in it, and it can accommodate 16 elements. And that it points to this thing that has not 2 or 4 or 8, but 16 elements in it. And I'm like, you know what, I did all this work, and I pushed on 14 elements right there, but I'd like to now pop off the top element.

When I do this stackPop, I have to pass in that. I have to pass in the address of an integer. So the stackPop call has a variable called elemAddr that points to my variable called top, and it relies on this variable to figure out where to write the element at position 13. It happens to reside right there. The memcopy says where do I write it? I write it at that address right there. You would replicate this byte pattern in the top space, this returns, and I print out top and that corresponds to the number 7,300 or something like that.

If I really wanted to change this void *, I don't change the void * anywhere, I just use it as sort of a social security number or an ID number on the integer. If I really wanted to change this you'd have to pass in a void * *. There are scenarios where that actually turns out to be what you need, but this is not one of them.

Student: For this pop, let's say that this function doesn't work, that it's popping the wrong stuff. How do we test to find out if we pop [inaudible]. Like, for instance, let's say you wanted to pop something that was an integer, and you popped [inaudible], and then what if there was something wrong with the resizing, can we set that in that we just pop to like null or something and then –

Instructor (Jerry Cain):You could. If you really want to exercise the implementation of the stack to make sure it's doing everything correctly, you'll write these things, you may have heard this, what are called unit tests. Which are usually implemented – there's two different types of unit tests, there's those that are written on the dot c end, which know about how everything works, and then those that are written as a client to make sure that it's behaving like you think it should. If that were the case, and you wanted to protect against that, you could write all these very simple tests to make sure that things are working.

Because in unit tests, as a client you might for loop over the integers one through a million. You might actually set the initial allocation length not to be four, but to be one, so that you get as many reallocations as possible. You could for loop and push the

number one through a million on top. In a different function, you could actually pop everything off until it's empty and make sure that they are descending from one million down to one. And if it fails, then you know that there's something wrong. If it succeeds, you never know that something is completely working, but you have pretty good evidence that it's probably pretty close if it isn't.

You can change the implementation to kind of make sure that all of the parts that are really risky, like the reallocation, and the

--, and the memcopy calls are working, one thing you could do, and Assignment 3 takes this approach a little bit, right here I use this doubling strategy. If I really wanted to test the reallocation business, I could actually do a ++ instead of a times two on the allocation size and to reallocate every single time. Not because you want it to work that way, but because you could just make sure that algorithmically everything else works regardless of how you resize them.

Now, another answer to your question, if I didn't take that approach with the unit test answer, is that when you're dealing with generics in C, and void *'s, and memcopy, and manual guests, you have to be that much more of an expert in a language to make sure that you don't make mistakes.

It's very easy. Think about it this way, I know you're not gonna believe this yet because you haven't coded in C that much, but Assignment 3 you will. I know how most of you program. You write the entire program and then you compile it. And you have 5,500 errors. And you get rid of them one by one, and it takes you like three days, and then it finally compiles. And you run it, and even if it doesn't quite run the way you want it to it rarely crashes. Most people don't even know what the word crash means until they get to 107. You will next week, trust me.

As far as C++ is concerned, because it's so much more strongly typed than C is, it is possible for you to write an entire program, to have it compile, because compilation does so much type checking for you in a C++ program that uses templates, it's quite possible that once it compiles that it actually works as expected. It doesn't happen very often but it's certainly possible.

In a C program, where you're using void *'s, and char *'s, and memcopy, and memmove, and bsearch, and all of these other functions you're gonna need to use for Assignment 3, it's actually very easy to get it to compile. Because it's like, oh, void *, I can take that, yep, that's fine. It just does it on all the variables, and so it compiles. And you're like, good, it wasn't three days, it was one day. And then you run it and it crashes because you did not deal with the raw exposed pointers properly. So that's what makes certainly the first assignment in pure C with dealing with void *'s difficult.

But there's an argument that can be made to say that it's very hard to get all this stuff right every single time you program. I've already told you that right here I forget this s-> elemSize probably one out of every two times I teach this example. That's because it's

very unnatural compared to the C++ way of allocating arrays to think in terms of raw bytes. And you think in terms of sizes of the figures, you see the pictures in your head as to how big things should be, so you just remember that number right there but you forget about this. If you forget this right here, it compiles, it runs. If you're dealing with integers then your array is one-fourth as big as it needs to be to store all the integers you want there.

You will learn this and feel it at 11:59 a week from Thursday. All these types of errors, because it's gonna compile and it's gonna run very often. But occasionally it's gonna crash and you're not gonna know why, and you're gonna say, oh, it's the compiler. It's not, it's your code.

Okay. So we will talk more on Wednesday.

[End of Audio]

Duration: 51 minutes