ProgrammingParadigms-Lecture05

Instructor (Jerry Cain):Hey, everyone, welcome. I don't have any handouts for you today. You're all crankin' on Assignment 1, which was intended to be very short through Sunday night. The first real assignment went out Wednesday. That's due next Thursday evening. And at least until the mid-term, I'm gonna establish this Wednesday to next Thursday schedule with all the assignments so there's some reliability as to how the workload ebbs and flows.

When I left you last time, I was probably about 60 percent of the way through my lsearch implementation. I'm trying to go from type specific to generic, but I'm trying to do that in the C language.

So this is what I wrote last time.

void *lsearch

And let's see if I can write the parameters out a little bit more neatly this time.

void *key

void *base int

m is the length of that array

int lm size is the size of the elements

And that's technically all the implementation that lsearch needs in order to figure out where the boundaries are between neighboring elements.

The fifth parameter is the one I want to focus on for the next 20 minutes. It has to have this as a prototype. I like the asterisk, we don't need it right there but I like to keep it there, and then I take two void *'s. I don't need to provide parameters names here because I'm not implementing this function here.

The basic algorithm for a linear search from front to back, that doesn't change. It's just the fact that we're trying to present an implementation that doesn't care about any specific one data type.

So I want to do this for int i = 0; i < mi++. With each iteration, I want to manually compute the address of the i'th element. I can certainly do that in terms of base, lm size is the quantum distance to move with each hop, and then i obviously tells me which element I'm interested in. Internally, I want to do this:

void * lm (address)

This is the thing that's going to be compared against that key right there to figure out whether or not we have a match.

This is equal to numerically:

base plus i times om size.

But we mentioned last time that this is strictly pointer arithmetic against a typeless pointer. No, I'm sorry, the pointer has a data type; it's a type void * so it doesn't know what it's pointing to. Several people have suggested, or asked, why they just didn't make default to normal mac when this is a void *. The specification of C just said I don't want to allow point arithmetic by default against a void *, because there was a clear rule for what point arithmetic means when this is strongly typed.

When it's weakly typed with a void *, very generic, I'm just pointing to anything, and I have no idea what. You can't do this, the hack, and it really is a hack, but it's a well-known hack, is to sedate base into behaving like a character pointer just long enough to actually get a number out of this expression.

Drag the base here, say you're pointing to characters. Do technically point arithmetic against the character pointer. This, as an expression, is an integer. It's technically multiplied by size of char, but that's one. So this ends up being a char * that happens to point to a boundary between the i minus 1th and the i'th element. I assigned it to a void *. You do not have to cast the overall thing to a void * if you don't want to because this is a more general pointer; it's willing to take on any pointer type.

If after you do this, you use the comparison function written by the client that knows how to compare the things that reside at these addresses, pass in a key, pass in a lm address. And if that comes back with a match of zero, then go ahead and return (I want to return the pointer), so go ahead and return lm address. This ends the entire four-loop. And if I get to the end and I have nothing to return, I'll just return null as a centinal that nothing worked out.

This replaces the double-equals that sits in between two integers from the integer version we wrote in the middle of last lecture. Double-equals between two strong types that are atomic, it knows how to do comparison. In almost all cases, it just does a bitwise comparison and can come out with a -1, or a +1, or a 0.

When you go generic on the C compiler, you have to say that I know how to compare the elements because I know, even though this is generic code, I know what type of array I'm searching. This code does not. So you have to pass in a little bit of a callback, or a hook, to tell the implementation how it should be comparing your elements.

This is easy to understand. It's easy to just look at this and understand what's going on because you know what linear search is; that's not the hard part. The hard part is getting

all the pointer math correct in the char * trick, and actually aligning these things up properly, and invoking the comparison function properly.

Using this as a client is at least as difficult as understanding this code. If I want to go, just think in terms of the int domain, and I have the following, I have intArray is equal to (this is a shorthand way of initializing an array) int size is equal to, I'll just hard code it as 6. If I want to search for the number seven, I actually have to do the following: number = 7. I have to set aside space for the key that I'm interested in because I have to pass the address of that thing as the very first element to the lsearch call. Does that make sense?

You know that this is laid out as an array of length 6. The 7 resides there. I'm passing that and that width and 6 to the lsearch routine. I'm hoping that it returns that right there. I want to find the place where the very first seven in the array resides.

int * found = (This is how I would call lsearch.) lsearch & of number. Array (No & is needed. There's an implicit & because it's really & of array of zero) pass in 6, pass in size of int. That at compile time evaluates to 4, at least in our world. And then I have to provide a comparison function. I want to write a comparison function that's dedicated to comparing integers.

So I'm gonna write that right now, int compare. Now, this will have to be defined as a function before I call this code right here, that I'm having to implement it afterwards. If it's the case that found equals equals null, then you're sad, otherwise you're happy. Does that make sense to people? Let me write the comparison function so we can understand why it has to take the form that it is, int cmp, if it's gonna actually compile and match this as a prototype, it absolutely has to take two void *'s and return an integer. That is the class of function that's accepted as the fifth parameter right there. You may ask, well, can I just actually write a comparison function that takes two int *'s and returns an int? And the answer is you could, but you'd have to cast it to be this type right here. It turns out that this is all pointers of the same size. You would pass them then as int *'s, and they would be absorbed as void *'s. But it's just a much better thing to do is to actually write the comparison function to match that prototype exactly. The implementation of that is a little clunky, but it doesn't surprise you, it just has a lot of asterisks involved.

void * lm 1

void * lm 2

Just because some – let's write the seven right here, this is the thing called number. On an arbitrary iteration, it may pass this int as the first argument to the comparison function. This right here is being invoked right there. It's gonna pass in the address of that one isolated seven right there every single time. The second parameter's gonna get that, and if it fails then that, and if it fails then that, etc., until it runs out of space. I have to return a -1, or a +1, or a 0 depending on whether they match or not. I also, because I am writing this function, specifically to make this call, this constrains the prototype to take two void *'s, but I know that they're really int *'s. So because I'm writing that code as a client, I can

reinterpret the void *'s to be the int *'s that they really are. So I will do this, int * ip 2, and I will just set it equal to lm 1 and lm 2. It turns out in a pure C compile you do not need to do a cast there, it just understands that the cast is implicit; it has to do it. So I have these local variable, ip 1 and ip 2, that not only point to this and that right there, but they actually understand them to be four-by quantities to be interpreted as integers. Does that make sense? So all I have to do is return *ip 1 - *ip 2. I'm relaxing a little bit on the return type, I'm letting zero meet a match. And of course, if that difference is zero than of the same number, -1 and +1, I could constrain it to be that. I just want it to really be a negative number or a positive number to reflect the delta between the two. Does that make sense to people? So if you understand this, great. If you understand this, even better. I'm sure most of you understand this even if it's the first time you've seen this type of code. Once we actually understand how all of this stuff works you're gonna be very, very happy. It's a little hard to understand the very first time you see it. But you have to recognize that this is not exactly the most elegant way to support generics, it's just the best that C, with it's specification that was more or less defined 35 years ago, can actually do. All the other languages you've ever heard of they are all so much younger that they've learned from C's mistakes and they have better solution for supporting generics. There are some plus's to this. It's very fast. You only use one copy of the code, ever, to do all of your lsearching. The template approach, it's more type safe. You get more information and compile time, but you get code bloat because you've got one instance of that lsearch algorithm for every single data type you ever searched for. Does that make sense? It's easier to get this right because you're not dealing with atomic types that are themselves pointers. We have integers right here. This gets a lot more complicated when you start dealing with the problem of lsearching an array of C-strings. Okay. So you're going to have an array of char *'s, and you're gonna have to search for a particular char * to see whether or not you have a match or not. These are the int boards. I want to deal with this setup right here. I have a small array 1, 2, 3, 4. And let's say I have an array of C-strings. Let's just assume that it's initialized this way. And I have an array of five little notes there. And I want to search the array using lsearch for an E-flat. So here's my key that I'm searching for; it points to an E-flat. I should emphasize that these are really character arrays that are null-terminated. Same thing with this. This right here is a character, character, character, character. This is a character array. That means that's a char *; char *, char *, char *. The address of the array right there, the arrow I've just drawn in, is technically of type char * *. How is lsearch gonna absorb that? It's going to absorb it as a void *. The only way it's gonna be able to compute that address, and that address, and that address as part of the linear search is because we're also gonna communicate the size of a char *, so it can manually compute the addresses of all those boundaries. The comparison function that needs to be written in order to do this has to be willing to accept addresses of that type and that type right there. This is where things can get confusing because you can kinda drift back and say, well, everything's a pointer so why does it matter that I pass in this as opposed to this? You're gonna see, when we write the comparison function, that the number of hops from the tail of the arrow that's passed in really matters. If lsearch passes this type of pointer to your comparison function then you really are two hops away from the actual characters that are compared to one another. Does that make sense to people? You may ask, well, why doesn't the comparison function just pass these pointers in? The answer is that lsearch has no idea that those

things are really pointers. The only thing it knows is if they happen to be for four-by-fourth of information. Make sense? Let me declare this: char * notes array is equal to, I'll write them as string constants, A-flat, F-sharp, B, then G-flat, and then an isolated D. I can talk about how these things are stored in a little bit. They're not in the heap, they're actually global variables that happen to be constant. It's like normal global variables, except they happen to be character arrays that reside up there, and these are replaced at load time with the base addresses of the A, F and the D.

char * (favorite note), as if I have a favorite note, it is E-flat.

Let me be very clear about this picture again – actually, let me draw it again; favorite note points to E-flat. The actual array happens to be of length 5. It points to strings A-flat, F-sharp, B, G-flat, and D. It's a cleaner picture. I want to search the array for my favorite note E-flat. The way you have to do this:

char * * (found)

Now, that enough is a headache. To understand why it's a char * * as opposed to a char *. But we'll get to that in a second. This is the way you would call lsearch. I have to pass in the address of my favorite note (I don't have to but I'm going to, I'll explain why in a second), & favorite note. I'm gonna pass in notes, that's the name of the array. Think about the data type of notes. Notes is the & of the zeroth element. Since the zeroth element is a char *, it's the base address of that capital A. Note is synonymous with that value right there. So even though it's being absorbed by lsearch as a void *, because it was written generically so that's the best it can do, we know that it's really a char * *. I have five of these notes pass in size of char *. Why char * as opposed to char * *? Because I'm interested in the actual width of these boxes so that lsearch can actually compute the boundaries between elements. And then, finally, with capital S and capital C, I'm just contriving the name of some function called StrCmp. Where actually there's two versions of StrComp, the one I'm writing and the one that's built into the C library, but the one that's in the C library doesn't have capital letters there. The reason I'm passing in the & here is because I want the true data type of the key, that's held by lsearch, to really be of the same type as all the pointers that are computed manually as part of the lsearch implementation. Does that make sense? If I know that this, and that, and that, and that, and that are all really char * *'s, it just makes life a little bit easier, if you have some symmetry inside code that's otherwise very complicated, to make sure that the key that's being compared against those five arrows is of the same data type. It doesn't have to be. I'll get to that in a second. But I'm writing it this way. So I pass in the & of favorite notes. So I get the & of the tail of that arrow that points to E-flat, two hops away from the capital E. I have to write the StrComp function. Even though lsearch returns a void *, it either returns a null, which we can check just like we did up there, or it returns one of these five arrows. Now, because E-flat isn't in there it's gonna return null. But if I had asked for the matching pointer to a G-flat, it would return that. I know that they're really char *'s in here. Lsearch doesn't but I do. So when I know it's returning this type of pointer, I know that it's truly of type pointer to char * or char * *. Make sense?

Student:

[Inaudible] the same as [inaudible] char * *?

Instructor (Jerry Cain):Yeah. The question is the size of char * the same as the size of char * *? The answer is, yes, because all pointers, at least in our world, are the same byte, and they're always the same size in any given system, and any given executable. You asked, I'm sure, just to be clear that they're both the same size, but you really do want this for readability purposes to be the true data type that's held inside the boxes of the array.

I could, if I wanted to, put 17 *'s there, and it would still work. That doesn't mean the code is the best way we could write it. Does that make sense?

Student:[Inaudible?]

Instructor (Jerry Cain):You don't have to. Right now, just for symmetry purposes, I'm making sure that the key and the addresses of all the elements in the array are of the same true type. I'll explain how we didn't have to bother with the & right here. You only can get away with that if you really understand what's going on. And I'm just not assuming that that's the case in the first 15 minutes of the example. But after I write it the one way, I'll explain how we could have gotten rid of that &. Okay. Let me get rid of these asterisks.

Okay. So I have to write StrCmp. I'm gonna do it over on this board.

int StrCmp takes two void *'s. I'll come up with better names this time, void * vp 2. The first one is always gonna be that address right there because that's what I passed in, & of favorite note. I know it's actually of type char **. On an arbitrary iteration, it might pass in the address of that right there.

So now that I've caused the implementation of lsearch, that right there, to just momentarily jump back to my type-safe code, the signature isn't type safe, but the code that's inside can become type-safe if I actually cast things properly. So I'm gonna go ahead and do this:

char * s1 (for string 1) is equal to *char * * vp1, *char * * vp 2

Now, why does that look the way it does. I'm casting vp1 and vp2 to be of the type that I know that they really are, this type and that type right there. They're two hops away from bonafide characters. After I do that, I dereference them once so that this as a value, and maybe this as a value are sitting in local variables called s1 and s2.

The reason I like that is because there is a built-in function as part of the clib that is completely in tune with the fact that the notion of a string is supported as character arrays that happen to be null-terminated, and that we pass around to those strings by address of

the first character. This is the address of the capital E. this is the address of the capital G right there.

What I can do is I can pass the buck to this built-in function with a lower case s and a lower case c, s1, s2. It takes two char *'s, and it knows how to do the booforce comparison of characters one after another, as long as they match continues. If it ever finds two characters that don't match then it knows that it can't return 0, it just returns the difference between the two ASCII values of the non-matching characters. That even applies if you hit a backslash 0 and 1 before you hit a backslash 0 and the other one. The delta is still what's returned. Does that make sense to people?

Student:[Inaudible] char *?

Instructor (Jerry Cain):Why didn't I just cast it to be a char *?

Student: Yeah.

Instructor (Jerry Cain):That's actually the question everybody asks right at this minute, the last 18 times I've taught the lecture.

So this right here is saying – is recognizing – I'm recognizing the vp1 that's being passed to me is really two hops away from actual characters. So that's why the double * is really the right thing there. And then I want to get two values that are just one hop away from the real data because that's what the built-in StrComp wants. StrComp, just like my intCompare function, it returns 0, -1 or +1, so it happens to return the value that I'm interested in.

So you're questioning why a double * here and then dereference once when I might be able to just get rid of those two things and put char *? Is that what you're asking?

Student: Yeah.

Instructor (Jerry Cain):Okay. The problem is is that * in front of the open paren, as opposed to the other two *'s on each line, that really is an instruction to hop forward once in memory and do a dereference. If I pass this as vp2, I say that you're not pointing to a generic pointer you're actually pointing to a char *. That's what the char * * cast does. And then when I dereference it once I do that. Given the way I've set up the call right there, if I were to do this, this would take this right here and it would assume that the actual material right there are actual characters. Does that make sense?

Student: Actually, I understand that [inaudible].

Instructor (Jerry Cain):This right here?

Student:[Inaudible.]

Instructor (Jerry Cain):Well, that's actually the part that does the hop and goes from here to there right there. You're just dereferencing a pointer. Does that make sense?

Student: Yeah.

Instructor (Jerry Cain):Was there another question flying up somewhere?

Student: Why [inaudible] char * * [inaudible]?

Instructor (Jerry Cain):Well, what's the alternative?

Student: Just like referencing the [inaudible].

Instructor (Jerry Cain):You actually could do that. That's where it confuses matters a little bit. But a void * you can't dereference because it doesn't know what it's pointing to. A void * * knows that it's pointing to a void *. Does that make sense to people?

I actually want to bring it into the char * domain as quickly as possible because then I really know sooner than later that I'm actually dealing with strings. Otherwise, I'm just leveraging off of my understanding of memory in a way that might not be clear to the person reading the code. Other questions?

Now, somebody asked about this right here. My implementation of lsearch up here, it's very careful to pass in key as the first of the two parameters to every call-up comparison function. Does that make sense?

Somebody asked what happens if I forget the & there. Well, my callback function still interprets whatever pointer is passed in as a char * *, so rather than this being passed as the first argument to the comparison function every time, and pass this in, it would still do a dereference after it cast this to be a char * *. So that would mean momentarily it's pretending that the E and B, and the backslash 0, and the mystery character that's right there, that that actually represents a char *, and then it would pass that to StrComp. That would not be good because it would jump to the E-flat mystery address in memory and just assume that there are really characters there.

However, not that I like this for this example, but if you know what you're doing and you want to pass this in right here, you just don't want to deal with the overhead of a dereference when you know you don't need to, you could pass this in. And you could recognize that the first argument that's being passed in is actually one hop away from the characters, and the second one is actually two hops away from the characters.

Student:[Inaudible.]

Instructor (Jerry Cain):Well, I can say the way I wrote it first is the way it's typically written. Because of the symmetry, I think coders, I don't know if they like to see it, I think they're just in the habit of only dealing with comparison functions that really deal

with the same incoming data type. And that's not the case if one's a char * for real and one's a char * * for real.

So it is more common for you to put an & right there, and to do this just so that the first line and the second line kinda have the same structure.

Now, for Assignment 2 search certainly comes up. As opposed to all of these examples, you know that there's some sordid flavor to the arrays that you're searching there. If you haven't read Assignment 2, again, I'll try to be as generic as possible in my description. But you basically have the opportunity to binary search as opposed to linear search for Assignment 2.

There's a built in function called bsearch. It turns out that there's a built-in function called lsearch, as well. It's not technically standard, but almost all compilers provide it, at least on UNIX systems. I'm gonna want you to use the generic bsearch algorithm, which has more or less the same prototype as lsearch right here, that's why I chose the prototype the way I did there, and it just does a generic binary search. You can implement it again yourself. If you already did then don't go back and call bsearch that's built-in. But I'd actually prefer you to use the built-in just so you learn how to use it.

This is the prototype for that built-in: void * is the return type. It's called bsearch, or naturally binary search. It takes a void * called key, it takes a void * called base, it takes an int, I think it's called len for length. I actually like n better, though, n always means size of an array. Int lm size, and then it takes the comparison function that takes two void *'s. The algorithm – in many ways the pointer mechanics are exactly the same as they are up there, the only part that's different is that it kinda does this binary search to figure out what index to probe next. It assumes that the data is in sorted order.

Now, I am going to say this, and you have to recognize it even though it doesn't sound very deep and insightful, it is. If you want to do the bsearch use this function for Assignment 2, and you want to do it as elegantly as possible. You have to recognize, kind of in sync with what I did over here, when I erased the & right here, you can pass in the address of anything you want to provided the comparison function knows that the first argument is gonna be the address of that something. Does that make sense to people?

With the & I pass in a char * *, without it I pass in a char *. I could have constructed a record and put four pieces of information in there, passed in the & of it, and then I could have cast the address that comes in as the first argument to be the address of that type of struct. The reason I'm saying that is because you're gonna want to do exactly that for Assignment 2. You're gonna need more than one piece of information to be available to the implementation of what you pass in right here.

As far as this is concerned, I've never said this in lecture before, but I'm glad I'm remembering right now, it has to truly be an actual function. CS106b and 106x, I don't want to say they're careless about, but they're just not concerned about it at the time. They use the word function everywhere for any block of code that takes parameters.

When I say function, I'm talking about this object-oriented-less unit, which is just some block of code that gets called as a function that has no object or class declaration around it.

When I'm talking about the type of number functions or functions that are inside classes, I don't refer to them as functions, I refer to them as methods. The difference between a function and a method, they look very similar, except that methods actually have the address of the relevant object lying around as this invisible parameter via this invisible parameter called this.

The type of function that gets passed right here has to be either a global function that has nothing to do with the class or it has to be a method inside a class that's declared as static. Which means that it does not have any this pointer passed around on your behalf behind the scenes.

I'll probably send them an email just about that one point. Because if there are two or three problems that everybody has with Assignment 2, one of them is related to this thing right here. Do you guys know about the this pointer from 106b and 106x? I think they actually used this even more in 106a, when they talked about Java, and it seems to come up more there.

C++ methods, those number functions that are defined in classes, normally pass around the address of the receiving object via an invisible parameter called this. And if you need to, you don't very often have to, but if you need to you can actually refer to the keyword this inside the implementation of any method, and it just evaluates to the address of the object that's being manipulated. That's what makes a method different than a regular function. Regular functions have nothing to do with objects so there's no invisible this pointer being passed around. You have to pass one of those object-oriented-less normal functions, or the name of one, as the fifth primary to bsearch.

Student: Why is it that the comp function [inaudible)] behind before.

Instructor (Jerry Cain):This right here?

Student: Yeah.

Instructor (Jerry Cain):Because these parenthesis were here, it's clear syntactically that it has to be a function pointer. And until about four years ago the asterisk inside was always required, and now it's just not. Because just the lexors and the [inaudible] know how to just decide if this is a function pointer type.

I like the pointer there, for various reasons, just because that's how I used them for the first 17 years I coded in C. And then someone went and changed it on me, and I'm like, I don't care, I want to use it the old way. That's a very C way of looking at it, too. There's nothing modern about C, so you shouldn't adopt any of it to modernisms. Any other questions at all?

There are a billion little generic algorithms I could write, but I don't want to focus on these. You now have all the material I think you need to really make progress this weekend on Assignment 2 if you want to. Assignment 2 is definitely a jump up from Assignment 1. Assignment 1 is intended to be all about UNIX, and just whenever you had time to get to it just to learn the UNIX that's necessary and then code up 20 lines of code to get RSG running. This is the one that really has some real C-isms that are required for the first half of the program. The second half, where you do the search, that's very C++-ish. Cubes and stacks and all that kind of stuff you've seen that before.

What I want to do now is I want to transition from generic algorithms to generic data structures. And you probably have more practice with generics and templates in C++ with the vector, and the q, and the map, and the stack, and all of those things. I think more often than not, people program in C++ as if it's C that happens to have objects, and they use the vector and they use the map. They don't use the ones from 106, they use the ones from the actual built-in STL library. A lot of people code procedurally, and write C functions, and they happen to incidentally use the vector and the map as data structures.

What I want to do is I want to write the same exact thing, support the same type of functionality in some C generics, recognizing that we don't have references, and we don't have templates, we don't even have classes. So we have to do the best we can to imitate the modern functionality that's offered by C++ and Java, and their templates, using C that has none of it.

So what I want to do is I want to slow down a little bit, and I want to implement a stack data structure. I want to make it int specific just so we have a clear idea as to how the generic should be implemented. But I'm just gonna go up front and say, we're gonna just implement everything in terms of int 's so there's no void * business yet.

Just as there in C++, you'll normally be very aggressive about separating behavior and implementation using the dot-h and the dot-CC scheme. But if you're a pure C you don't use dot-cc as in extension you use dot-C so you know that the file contains pure C code as opposed to C++ code.

So what I want to write here is a stacked out h file, and this is how I'm gonna do it. There's several ways to do it in C, but I want to imitate the way you're used to it from C++ as much as possible.

There's no class keyword in C, but there is the struct. We're gonna use that. There's no const, there's no public, and there's no private. Our compiler actually supports const, but there's certainly no public and there's certainly no private. So what I want to do is I want to come as close to the definition of a class right here as possible using just C syntax. And this is how you do that:

typedef struct (The typedef keyword is required in C; it's not required in C++).

And then I want to do the following:

int * lm's

int logical (length)

int allocative (length)

And that is it. I want to call this thing a stack.

Now, in the dot-h file, when I define the struct right there, technically all three fields are exposed so they're implicitly public. Documentation above the dot-h, at least in Assignment 3 when we start doing this type of stuff, it's very clear that we're just exposing these three fields for convenience so people can actually declare stacks as local variables, and the compiler knows that they're 12 bytes tall but that you should not manipulate these three things at all.

You should just rely on the functions, not methods, but functions right here to manipulate them. And just take this, accept for your ability to declare the stack and that you know that it has three fields inside. Think of any struct as a black box where you just aren't afraid to manipulate the 12 bytes that are inside.

I want to write a constructor function. I want to write this destructor, or disposal function, and then I want to write an is empty function, a pop function, a push function, things like that. So here's the prototype of the first thing I'm interested in:

void * stack (new)

All I'm gonna do is I'm gonna pass in or expect the address of some stack that's already been allocated.

We were talking about the this pointer before. You know how when you call a constructor in a class it has access to that this pointer, it's because it's passed in as like the -1'th parameter, or this invisible parameter before everything else. All we're doing is we're being very explicit about the fact that the address of the receiving structure is being passed in as the zeroth argument. We have to because that's what C allows us to do.

I also have this function stack dispose. I want to identify the address of the stack structure that should be disposed. This is gonna be a dynamically allocated array that's not perfectly sized. So I want to keep track of how much space I have and how much of it I'm using. I also want these methods. Let's forget about the is empty and the def, let's just do it with the real functions.

Void stack push

What stack am I pushing onto? The one identified by address right there. What integer's getting pressed? This one. And actually we'll go with an int right here, stack pop. Which

stack am I popping off of? The one that's identified by address right there. I just want to be concerned with those things right here.

I don't know that I'm gonna be able to implement very much because I only have about nine minutes left, but I can certainly, without code, just like pictures that serves a pseudo code, just give you some sense as to how things are gonna work.

The allocation of a stack, when you do this, conceptually all I want to happen is for me to get space for one of these things right here. That means that this, as a picture, is gonna be set aside. And you know, based on what we've talked about in the past, that it's 12 bytes if the lm field is at the bottom, and that the two integers are stacked on top of it. But as far as the declaration is concerned, it doesn't actually clear these out, or zero them out like Java does, it just inherits whatever bits happen to reside in the 12 bytes that are overlaid by this new variable.

It's when I call stack new that I pass in the address of this. Why does that work, and why do we know it can work? Because we identify the location of my question-mark-holding stack pass into a block of code that we're gonna allow to actually manipulate these three fields. And I'm going to logically do the following:

I'm gonna take the raw space, that's set up this way. I'm gonna set it's length to be zero. I'm gonna make space for four elements. And I'm gonna store the address of a dynamically allocated array of integers, where these question marks are right here, and initialize the thing that way. That means that that number can be used not only to store the effective depth of the stack, but it can also identify the index where I'd like to place the next integer to be pushed.

So because I'm pre-allocating space for four elements, that means that this is a function. It's gonna be able to run very, very quickly for the very first calls. And it's only when I actually push a fifth element, that I detect that allocated space has been saturated, that I have to recover and panic a little bit and say, oh, I have to put these things somewhere else. That it'll actually go and allocate another array that's twice as big, and move everything over, and then carry on as if the array were of length 8 instead of 4 all along.

You've done this very type of thing algorithmically. At least you've seen it with the C++ implementation of templates, and at least just these type of data structures from 106b and 106x. I'm just doing this because I want to be able to start talking about the same implementation with int 's. Using 107 terminology we're gonna be dealing with arrays. You can imagine that when we go generic this is still gonna be an array, just like the arrays passed to lsearch and bsearch are, but we're gonna have to manually compute the insertion index to house the next push call, or to accommodate the next push call, and do the same thing for hop.

I do this [inaudible] int i = 0; i < 5, i++. I want to go ahead and I want to do a stack push. Which stack? The one at that address, and I just want to pass in i. Just draw the picture as to how everything's updated. And then right here, rather than dealing with the pop

problem, which is actually not anymore difficult than the push problem, I just want to go ahead and stack dispose & of s.

So from a picture standpoint, the very first iteration of this thing is gonna push a zero onto the stack, it's gonna push it at that index. So I'm gonna put a zero right there, and put a 1 right there. It's that 1 that more or less marks the implicit boundary between what's in use and what's not in use. Make sense?

Next several iterations succeed in sending that to 2 after there's a 1 there, and 3 to put a 2 there. It makes this a 4, puts a 3 right there. It detects that now as the boundary between what's in use and what's not in use. You could reallocate right here if you wanted to. I wouldn't do it yet, I would only do it when you absolutely need to on the very fifth iteration of this thing. So what has to happen is that on the very last iteration here I have to do that little panic thing, where I say, I don't have space for the 4. So I have to go and allocate space for everything.

So I'm gonna use this doubling strategy, where I've gotta set aside space for eight elements. I copy over all the data that was already here. I free this. Get this to point to this space as if it were the original figure I've allocated. Forget about that smaller house, I've moved into a bigger house and I hated the older house. And then I can finally put down the 4 and make this a 5 and make this an 8. So that's the generic algorithm we're gonna be following for this code right here.

Now, I do have a few minutes. Let me implement stack new and stack disposed. And then I'll come back and I'll deal with stack push the beginning of Monday. I just want to go ahead and put the dot-h here and have its dot-c profile right to its right.

I want to implement stack new. So take a stack address, just like that, recognize that s is a local variable. It has to make the assumption that it's pointing to this 12-byte figure of question marks that is that tall right there.

So what I want to do is I want to go in. I want to s arrow logical n = zero. I want to do s arrow alloc len = 4, and then I want to do the following: I want to do s arrow lm's = (this is a function you have not seen before) malloc times 4 times size of int . Now, if I tell you that this is dynamic memory allocation you're not gonna be surprised by that because the word alloc, the substring alloc, comes up in the function. This is C's earlier solution to the operator new solution. We don't have new and delete in pure C, we have this raw memory allocator called malloc.

Operator new takes account and an implicit data type, because you actually say new into 4 or new double of 20. You don't do that in C. Not that we should be impressed with the idea, but the way malloc works is it expects one argument to be the raw number of bytes that you need for whatever array or whatever structure you're building. And if I want space for four integers that's certainly in sync with this line, where I'm saying I'm allocating four of them, you have to do four times the size of the figure, it goes and searches for a blob in heap, that's 16 bytes wide, and it returns the address of it.

There is some value in actually doing this. You've seen the assert function in the assignment starter code, or Assignment 1. There is this function called assert. It's actually not a function it's actually a macro. There's this thing you can invoke called assert in the code, which takes a boolean value, takes something that functions as a test. It effectively becomes a no op if this test is true, but if this is false assert actually ends the program and tells you what line the program ended at. It tells you the file number containing and the line number of the assert that broke.

The idea here is that you want to assert the truth of some condition, or assert that some condition is being met, before you carry forward. Because if I don't put this here and malloc failed, it couldn't find 16 bytes (That wouldn't happen but just assume it could) and it returned null, you don't want to allow the program to run for 44 more seconds for it to crash because you de-referenced a null pointer somewhere. You just don't want to dereference a null pointer because it's not a legitimate pointer it's a centinal meaning failure. So you don't want to dereference failure because that amounts to more failure.

And then you'll get something, while the program is running, called a seg fault or a bus error. And I'm sure some of you have seen them. Those are things you don't want to see. You'd rather see an assert, where it tells you what line number was the problem as opposed to a seg fault, which just says, I'm a seg fault, and notice your program stops.

This can be stripped out very easily so that it doesn't exist in production code. You actually don't want this failing if a customer is using this code because then it makes it clear that your code broke as opposed to their code. This can be very easily stripped out at compile time without actually changing the code.

So when we come back on Monday I'll finish the rest of these three and then we will go generic on you.

[End of Audio]

Duration: 52 minutes