ProgrammingParadigms-Lecture04

Instructor (Jerry Cain):Hey, hey everyone. Welcome. You made it through a week of 107. I have two handouts for you today, although I really only have one. I have one fresh handout, and I also have hard copies of the discussion session handout from yesterday, which I know not everybody can go. So if you need a hard copy of that and don't want to print it out yourself, then come grab a copy before you leave. When I left you last time, I had gotten through the implementation of swap that was specific to ints. So I want to make a few points about that and then go generic on you by implementing the C version of what we would do in C++ using templates.

This is more or less the code I wrote for you last time. The idea being that AP and BP actually address in some mysterious space. They know the address of it, but they don't know what the source of it is. Whether it's the heap or subfunction call or whatever, but algorithmically what happens is that the two integers in those boxes are effectively exchanged. Now the 106A or 106B way of saying this, in spite of the fact that it uses pointers, is that it actually rotates the integers.

The 107 spin on this, which I think is more helpful for the code we are going to write in a second, is that, oh, I don't really care that they're integers as long as I exchange the representations for those things – the 4-byte representations for these two integers. Then when we go back to the code that calls this, it will notice that two of its integer variables, whether they are embedded inside a raise or struct or they are two stand-alone integers – their bit patterns will have been exchanged, their representation is being exchanged so that when they look at those they'll be each other's integers. Okay, does that make sense to people? Yes? No? Okay, it did not or did? We got a nod.

The reason I say this is because the implementation here – and I'm going to frame this with a 107 bent on it. This declares a 4-byte figure, and this assignment replicates the four bytes held by this box in that box right there. It knows that we are dealing with a 4-byte figure because this and this and temp are all typed in a way that's related to an integer. Okay? This does the same thing, takes a bit pattern right here and replicates it in the space addressed by AP, and then finally remembers what used to be pointed to by AP and puts that in what's pointed to by BP. Okay, so it's really a bit pattern rotation.

There is an implicit knowledge of the number of bytes that are being moved around because ints are just understood even in compile-time to be 4-byte figures. If I want to use this function to swap doubles, I am not going to be able to do it. If I want to be able to swap two structs or two classes, I am not going to be able to do it.

What I want to do is I want to write a version of swap that can exchange – I'm gonna design it to exchange two arbitrarily sized figures. I'm sorry – the two figures themselves will be the same size, but I don't want to constrain it to be 4 bytes. So what I want to do is I want to write this as a function void swap, rather than accepting an int * and requiring that I get the address of an integer or something that's posing as an integer. I want to be able to pass in an arbitrary address here, and I don't want to constrain it to point to any one type.

The way you do that in Pure C and even in C++ technically, but in Pure C is to write down a generic pointer type. That is a type void *. Now, that doesn't mean that it points to nothing; it just means that it points to something that doesn't have any type information about it. Okay? And I'll put down VP1. The second argument will be VP2. The set-up here is that VP1 and VP2 are addressing some things that begin at the addresses that are stored there.

Now I draw them as L's as opposed to rectangles because I don't know how wide they are. Does that make sense to people? Okay. And it's really a generic address. It's just an arbitrary location in memory. There may be a character, there may be a short, there may be a Boolean and there may be an unsigned long. There may be a struct Fraction or a struct student. We just don't know.

Let me make the mistake of closing this off and showing you what problems we run into. If you try to do this – I'm not trying to be funny here, but if you try to do something like this – your heart is in the right place, but this is just plagued with issues. There is one quite clear problem with this and there is one slightly more subtle problem with this. You cannot declare a variable called temp to be a type of void.

Okay, that's just a return type for functions. That just states that there is nothing to be returned. You can pass it in void as a lone argument to a function or a method to say that we're not expecting anything or you can use void in the contents of void * to mean generic pointer. You cannot declare temp to be a void, okay?

The more subtle problem here is that you are not allowed to dereference a void *. And you may be like, "Well, why not?" And the answer is it doesn't know how many bytes to go out and embrace as part of the identification process, okay. "Do I go out and do I deal with a 1-byte figure, a 2-byte figure, or a 4-byte figure?" There is no type information about this, so it doesn't know whether it is 4, 16 or 128 bytes.

Does that make sense, people? It has no size information about the thing being addressed at all. So the official thing to do, recognizing that we still want to rotate bit patterns, is to expect a third argument. Int called size where size is supposed to be explicitly stated as the number of bytes making up the figures being swapped.

So at least it has more information than it had before. It actually doesn't really care whether they're 4-byte integers or 4-byte floats or a struct with two shorts inside. As long as I exchange the 4-byte bit patterns, I am effectively swapping the values. This is how you do it – (char) (buffer) (size). Our version of GCC and G++ allows you to declare arrays with a size that depend on a parameter. So this might seem weird that I'm declaring a character buffer, but it isn't really a character buffer in the C-string sense. I'm really just setting aside enough space to hold size bytes so it can function as temp does in that block of code up there.

I don’t care to interpret buffer as a string. I just want it to be this little storage unit where I can copy something. Remember how last time I went over this function called strcpy that knew how to copy bytes from one location to another location and it kept on copying until it found a \0 and it copied the \0 as well? There is a more generic version of that that is not dedicated to characters.

There is a function called memcpy. What that’s taken to do – it’s like strcpy, except it doesn’t pay attention to \0, so you have to explicitly tell it how many bytes to copy to its memory location. If I write buffer there and I write VP1 there, that’s an instruction to keep copying bytes, you can think about it copying bytes one by one; one byte after another to the space addressed by this right here. Okay?

This is the source of those bytes. It doesn’t care about zero bytes. You may be copying 20 bytes of zeros; this is why you need the size parameter to be passed in so you know how many bytes should be copied. So before I finish this, let me just give you a sense as to what is happening here. Suppose this is in fact an 8-byte figure and this is the bit pattern that is right there. This declares something that is as wide as that. It's not to run the scale but I will just emphasize the fact that it is all characters.

This right here says, “Please copy stuff from that address into this address right here,” and it just does it byte by byte. It doesn't matter that they're not really characters. They are just bit patterns that are taken or digested; one byte at a time and the full bit pattern that’s right there is replicated in that perfectly sized space. Does that make sense?

Only in Java, it does; it doesn’t in C++, it’s only one byte. Yeah. The memcpy right here basically does the equivalent of that first line up there. It just took two lines here. Then what I can do is I can do a memcpy into the space addressed by VP1 from the space that is addressed by VP2 and copy the same number of bytes. That takes that right there and, as a bit pattern, replicates it over this space.

And then finally, I do this – copy to the space VP2, the stuff that was stored in buffer, and I get that then. So it achieves the same byte pattern rotation that you see in that very type specific version up there; it just does it generically. Okay, does that sit well with everybody? Now you may look at this and say, “It’s kind of ugly.” It is kind of ugly. There are actually a lot of problems with this.

This right here, that declaration of an array, is supported by our version of a compiler. True anti-C that’s compatible with all compilers actually doesn’t allow you to put anything other than a constant inside. I don’t mind if you do this. You can use the compiler that you have. But the real implementation to probably dynamically allocate a block that is that number of bytes, move it, use it as a temp even though you are copying to the heap as opposed to the stack, and then you get to free it at the end. So most of the energy is invested in the dynamic allocation and de-allocation of a buffer or a temp space. Okay? Do you guys understand this function right here?

Well you wouldn’t – you would call malloc, which is like OperatorNew from C++ and you would – I’ll talk about malloc when we get there, but it’s just the C equivalent of OperatorNew. I just like this version better because it’s a little cleaner and I want to talk about memcpy more than I want to talk about malloc. The thing about this – you say, “Okay, well, that’s great. I guess I have to deal with the void *s but it’s not that bad.” The problem is that lots and lots of things can be disguised as void *s.

Let me make the proper call here. If I go ahead and declare (int, x = 17) and (Y = 37) and I do this (* of X), (* of Y) and I pass in – you could pass in the number four, but that’s not a cross platform solution you want – not the size of four. That would actually return four. That’s the way the client has to interact with this generic function right here. Identify where those two ints are; the swap implementation doesn’t care that they’re ints, it just cares that they are 4-bytes wide, so it does the right number of byte rotations as far as these three calls. Does that make sense to people? Okay. The problem comes if you try to do something like this (double) – actually, this is not a great example.

Let me just do this (d = pi) (e = e). Just pretend that that makes sense. And I want to make the call for this. You do this. And the same code works. Let me frame some plusses of this right here. The same code gets used for both of those calls right there, okay? It emphasizes the fact that it’s this generic byte rotator. Think about what you’d have to do in C++. I know you probably know you’d use templates in C++.

The one perk of this over templates is that just this code gets compiled and the same assembly code that corresponds to this right here gets used for both calls. When you deal with templates, there are many plusses of templates, but it expands a compilation or a call to swap of int or swap of double in a template setting, actually expands two independent versions of the same code and compiles them in the int specific domain or the double int specific domain.

Do you understand what I mean when I say that? Okay. That’s not a tragedy if you’re only calling swap twice but if you call swap in a very large code base, you call swap fifty different ways with fifty different data types you get fifty different copies of the same code in your executable.

Okay, one is set up to deal with chars, one’s set up to deal with the shorts, one’s set up to deal with the ints, etcetera. This is very lean and economical in the way that it deals with the swapping process. The problems – I actually say, “I’m not trying to illustrate this as the best solution; this is just what C has to offer.” The problem is there are so many mistakes that can be made when you are dealing with a generic function like this.

Swap is pretty easy in the grand scheme of things but we’ll see in a second, that it’s actually easy to get the call wrong and for the compiler to tell you nothing at all, because it’s very easy to be a void *. Okay? You can pass in the address of a float, the address of a double and pass in 32 right here. It’s not going to work very nicely when you actually run it, but it will compile. Does that make sense to people?

So these void *s, particularly the cast we've been dealing with for the past two lectures, they kind of sedate the compiler enough so that it doesn't complain when it otherwise would have complained. Okay? That might be great to actually feel like you might be making progress towards your goal, but you really do want the compiler to edit and coach you as much as possible.

So to the extent that you use generics in C and cast in C, you are basically telling the compiler not do as much work for you and you are just risking more when you actually run the program. Okay? You wouldn't make this call, but just pretend you did. Suppose I do an int right here. I is equal to 44, and I do this, short s is equal to five, and logically, what I want to do is, I just say, "For various reasons, I need to view the different sizes but now I need the 5 and the 44 to logically exchange positions." And you do this. And you just pass in the smaller of the two sizes.

The memory set up here is i is that wide, it has a 44 inside. S has a 5 inside. The VP1 and the VP2 that accept these addresses. Even though this is really an int and that is really a short, VP1 and VP2 don't have that. It's like they don't have their typed contact lenses on or something. Okay, they just have the address itself and the only reason they know to access those two bytes and in this case, those two bytes is because we explicitly tell it how wide the figure is right there. Okay?

Now algorithmically follow the recipe right here. What is going to happen is it's going to take this 5 and write the bit copy for the 5 in the left half of (i) right there. It is going to take whatever happens to reside right there; those two bytes, and replicate it down there. Okay. On a big-endian system, it's going to put this 5 on the upper half of (i) and not clobber the 44, okay? And it's going to take all of the zeros that used to be here and put it right there. So as a result of this call right here, (i) would take on a value of 5 times 2 to the 16th plus 44 and (s) would become zero. Does that make sense to people? Okay.

It would be a little bit different on a little-endian system. It actually would kind of get it right on a little-endian system, okay? But it would be a complete miracle that it is.

This isn't a – it's actually – it depends on what you call a disaster. This will survive compilation because this is a generic address. This is a generic address and this is effectively an int. So the compiler says, "Are you calling the functions properly?" Yes, I'm getting two addresses and I'm getting a number. Okay, and then it just runs this code, it exchanges the bytes according to its own little recipe inside and then whatever side effect is achieved by that rotation of bytes, is what you see when you go and you print i and s out.

Absolutely, this recipe, because size of short was passed in. It doesn't even see the 44 and the other two bytes. It's just not in its jurisdiction. Okay.

Oh, that was a mistake. Sorry. This should have been doubled. This was intended to be a valid call. They are all valid calls actually, but only some of them work.

I just chose buffer, it doesn't have to be that. Yeah, I just chose it to emphasize the fact that it really is just this generic store of bytes. I could have called it temp; I just didn't.

Yeah, anything with very few exceptions, everything that's legal C code is legal C++ code. Just some things about type casting are a little bit different, that's it.

Yeah, absolutely. It would actually be worse. If I were to put the word int right here all it does is, it gives the swap function a wider pair of arms to go and grab 4-byte figures instead. So this as a byte pattern would be exchanged with that space as a byte pattern, okay? If it didn't crash and it ran, (s) would just take on whatever happened to be the left two bytes in the (i) figure.

Not using this right here. In that situation, you would have to write either an int short specific version of swap or you would have to allow for the possibility that you pass in two different sizes, one for each of the two void *s. It would be complicated. It is probably the case that you would not make any of these mistakes.

If you are dealing with atomic figures like doubles and ints and shorts, you are just probably not going to make a mistake like that. Okay, it's a little troubling that the language doesn't actually enforce the rules you want it to, but in this case, it really doesn't amount to much. There is an example where I think it does but it's gets to be more complicated and that's what I'm going to do next.

Well, you could, but then once you cast something, it forces it to evaluate it and so you wouldn't be passing in the address of s. You'd be passing in the address of the constant 5, and that doesn't make sense. Okay, you actually have to have storage associated with an address, so it has to correspond to the addresses of some variable.

You could if you wanted to, set like ints ss = s and then do a swap between (i) and (ss) and then after it was over, the set s = ss to whatever it turned out to be. You could do it that way. That's a weird band-aid to overcome or to use, when it's really the function that's the problem. You would just want to write an int specific version of swap if you really needed to do this. Okay? Let me deal with data types that are already pointers.

You could. There is usually not very much reason to use const here. Usually you only use const – I know you are seeing a lot of const(s) in Assignment One. Const is only generally used when there is some sharing of information going on. This function owns its own copy of size. Does that make sense?

Oh, I see what you are saying. No, this still has to be evaluated, it would have to actually be a constant; like a 40 or an 80 or something like that.

It doesn't have to do with the fact whether it is changeable. This is the fact that it's an expression that evaluates to an int, but it is not actually an int. I don't want to confuse matters. I think this is fine because we have a compiler that happens to like it. Some compilers might not, but this is just an easier way to write this function. Still dealing with this code right here –

Well, we can't – if we do that, then we are trying to dereference a void *. You want to identify the address of the house with all of the bytes in it. Okay and that's why you just let VP1 evaluate itself. If you try to dereference it, even if you can't dereference it, if you dereference it, then you actually lose access to the address and so you don't actually get access to all of the bytes that are there. Some compilers do let you dereference void *s, but I actually set up the warning so that you can't. Okay? Any other questions at all? Okay, yep. Go ahead.

Well, if you set the warnings properly it doesn't let you; it's a compiler error. I'm sorry; it's at least a warning in G++ by default. It just assumes that it is a 4-byte figure and it just deals with them as longs. But I want you to assume that void *s can't be dereferenced. Let me write this block of code right here; there are so many ways this can be messed up. I have a char * called husband and I set it equal to *strdup of Fred. I have a char * called wife equal to *strdup of Wilma. I'm calling strdup here because I want independent copies of these screens to exist on behalf of this little snippet of code I'm drawing right here.

Let me draw the state of memory right here. I have this thing called husband, I'll just put an "h" there to mean husband. I have this variable called wife, which just gets a "w" here, and then in the heap over here, I get space for Fred\0, I get space for Wilma and this points to that as a result of second strdup call, that points to the first one as a result of the first strdup call. And what I want to do is I want to exchange, just for a day, I want Fred to do all of Wilma's work and Wilma to do all of Fred's work. So we can have [inaudible] point to it as if it really existed.

So what I want to do is I basically want to exchange the two strings. I want the husband variable to be associated with the Wilma string and I want the wife variable to be associated with the Fred string. The correct way to call this, it's actually quite confusing. I want to call swap. A very reasonable question to ask here is whether you need an ampersand, because you say, "Okay, husband and wife are already pointers."

I'm going to write it the right way. I'm going to put address of husband, I'm going to put address of wife and I'm going to put size of char *. Now don't think too hard until I say a few more things. I actually want to exchange the two things that are held by the wife and the husband variables. Does that make sense?

When I wanted to exchange two ints, I passed int *s to swap. Okay? If I want to exchange two char *s or actually, I want to exchange things that are that many bytes wide. If I want to exchange char *s, I have to pass in the address of char *s to swap. Okay, that way it swaps these things right there. Okay, does that make sense? So this gets associated with VP1 in the swap implementation.

This right here gets associated with VP2 in the swap implementation. The size of char * is the size of this thing right here. That means I get a buffer of characters that is four bytes wide. Even though VP1 and VP2 recognize these addresses as generic and as void *s, I know that they are really char * *s. Okay?

The way this works is that this implementation forgets about the fact that there are these things over here and it forgets that these things really are char *s, it just rotates the bytes. So what happens is that this as a pattern, identifies the – I'm sorry, this as an address identifies that pattern as something that should be replicated, so it copies the address pattern right here and it happens to be interpreted that way.

Do you understand what I mean when I say that? This material is replicated right there. I draw it as a pointer, not because this knows it's a pointer because we know it's a pointer. Okay. This right here – I'm sorry, this material right there is replicated right there. And actually, technically that still points to Wilma. Okay?

And then finally this is updated to actually point to that right there. Okay, now it's a lot of arrows that are moving around but the "h" – the tales inside husband and wife are actually exchanged. Nothing happens to the capital F; nothing happens to capital W and all of the characters after them, they stay put. I just have Fred and Wilma as husband and wife exchange names.

Okay and there is some confusion in the matter because C-strings are just not as elegant as C++ strings and Java strings; they're very manual and exposed character arrays. But you have to exchange the char *s. Okay? The problem with this is that if you forget to do that right there, it will still compile and it will still execute. And it will actually still run, and it will do something. It will not crash. I promise you, okay?

Let me redraw everything. I won't be so careful with the drawings of Fred and Wilma. Here's Fred\0, here's Wilma\0 in memory, here's husband, here's wife, with an "h" and a "w." There's that and there's that. So I redrew it, its set-up and forget about the ampersands being there. What actually happens now – 4 gets passed to swap. So it's going to be rotating 4-byte figures. Okay? But I kind of mess up a little bit.

Even though I am passing in a char * here and a char * there and I'm passing in size of char * there, this address gets stored in VP1. This address gets stored in VP2. Without the asterisk, it doesn't back up one level. Okay? It actually gives you the address husband and wife actually evaluate to the tales of those pointers right there. So whatever the address of capital F and capital W are here, are stored there and there. They're evaluated and passed directly to the VP1 and VP2.

So swap is like okay, I got two addresses and I'm supposed to swap 4-byte figures. It goes and it actually copies, wilm there, and it leaves the a alone, and so those two strings would become – it would actually change the character strings without changing husband and wife itself to wilm and freda, okay? Does that make sense to people? Do you understand why it won't crash? It's actually accessing, even though it's not the material we wanted. it’s the material that’s under the jurisdiction of the code block. Okay, does that make sense?

If I were to do this, it would not care why you would do that, but think about the compiler was like, "Okay, I’m happy with that." “Yeah, I have two voids *s coming in,” one happens to be a char *. One happens to be a char **. What would happen is that the address that is stored in wife would be placed as the first four bytes of the fred string. Does that make sense to people? This right here would be replicated right there.

I can tell you, you can print it out, it's going to be something; it's not going to be a pretty string, but it's going to be a string that is no larger than four characters. It may be smaller because there may be a zero byte involved in the address. They would be random characters question marks, diamonds – whatever you see when you open one of those binary files accidentally. Okay? All those little numbers that don't happen to be letters of the alphabet or numbers or periods or what have you.

This right here, wilm right there, this is the problem. I’m sorry, the Fred that used to be there would actually be exchanged with this pointer so you would lay down Fred as a byte pattern in this thing that is going to normally be interpreted as an address. Okay? So that means that whenever 4-byte figure that corresponds to, if you pass wife to see out [inaudible] it's going to jump to the fred address in memory, which you certainly do not own if it doesn't crash because it's not inside the stack or the heap, which probably will be the case. But if it doesn't crash, it's just going to print out random characters that happen to reside at fred, interpreted as an address. Does that make sense? Okay.

I'm not encouraging you to write code like this, and even if it works, I'm not trying to get you to write it in as complicated a manner as possible. I'm just trying to communicate the things – when you write code and you kind of mess up on the type system, it’s not a tragedy because the compiler will usually tell you there is a problem, unless you're dealing with generics, right here. Okay? And then it says, “Okay, I’m just going to trust you because you told me that it was just a pointer. And I can't argue with just a pointer when they are pointers.” So you have to be oober careful about how you code when you're dealing with generics. Okay, very powerful, also very dangerous. Okay?

Because of the asymmetry, right here let me draw this a little bit more cleanly. When a husband points to fred\0, that’s 4 bytes, what I just underlined right there, right? When I pass an ampersand of w, this points to wilma\0. This is a 4-byte figure, but it turns out, that doesn't matter. Because the addresses I passed to swap are, this right there and that right there. That means that these four bytes will be exchanged with those four bytes. Okay?

And just to make it clear how ludicrous it is, that means that fred as a bit pattern will be placed f r e d, asked the value for f times 2 to the 24th, asked the value for r times 2 to the 16th; all be assembled and interpreted later on as a regular pointer, okay? Whatever this is, whatever the bit pattern is, it logically can be set up to point to capital W, although it is going to be interpreted not as a pointer but as four side-by-side characters. Does that make sense?

Okay. There are all types of mistakes that can happen here, you can include both ampersands and get it right. If you put a double * there, it actually works because all pointers are 4-bytes, at least on our systems. That doesn't mean it's the right way to do it. If you want you can put size of double *****, and it will work. Okay? But you really want to be clear about what you understand to really being exchanged. If you really know you are exchanging char *s by identifying two char **s, you should for clarity's sake, not just because you can get away with it, you should put size of char *, right there. Okay, does that sit well with everybody? Okay, good.

I want to graduate to a new example. Let me once again write a really simple function for you. Int – I'm just calling it L search. While I pass in an int, I'm going to call it a key int array int size, and I want this to just be a linear search from front to back of the array for the first instance of key in that array, and I want it to return the index of it or -1 if it can't be found.

So algorithmically, this is very 106A. What I want to do is this. I want to be prepared to exhaustively move over everything, but if along the way I happen to find array of i matching this key, I want to go ahead and return what (i) turned out to be, okay? If I get this far because I've exhaustively searched and found nothing to match, at the bottom, I return -1.

Now, I know that you think that that's all reasonable code, and you wish that all examples were like that, but they're not. The only reason I'm putting this up here is because I want to frame the implementation with the new vocabulary that's now accessible to us because of what we talked about for the last three days. This 4-loop, that right there, that right there, that's the same whether it’s int specific or generic.

There's a remarkable amount of stuff going on in that line right there. There's point arithmetic. There is basically an implied asterisk, that comes with the square brackets. Does that make sense? There is the double equals that actually does a bit wise comparison of the two 4-byte figures to figure out whether they are equal. Okay, does that make sense?

So if I want to go generic here and I don't want to engineer it to just deal with ints, that means that I have to pass in more information, more variables than I'm actually passing in right here. When that (i) is placed right there, let’s say it evaluates the three. It knows to go from the base address of the array, plus three times the size of int, right. Okay? And that's how it identifies the base address of the thing that should be compared to key on that particular iteration.

If I make this a void *, then all of a sudden, I lose the implicit point arithmetic that comes with array notation. In fact, you can't use array notation on a void * for the same reasons you can't dereference it. Okay, there is no size information that accompanies it. So this is what – and I also lose the ability to compare two integers. When I know they are integers, it’s enough to just look at the bit patterns in the space that we call ints.

When we don't know what they are, we don't necessarily know how to compare them. Maybe double equals works, okay? Probably not necessarily – certainly, it won't with strings. So if I want to write the generic version of this, I will have a better time doing it if I frame it in terms of a generic blob of memory. This is going to be the array that is linearly searched in a generic manner.

In order for me to advance from (i = 0) to (i = 1) and know where the 1th element begins, I'm going to have to pass in some size information about how big the elements are, so that I can manually compute what the addresses are. Does that make sense to people? Okay. I also am going to have to pass in a comparison function so that I know how to compare the key to the material that resides in what is just taken to be the (i) entry in the array. Okay, I can’t use double equals very easily.

So what I want to do is I want to write a function that returns the address within the array of the matching element. Just a generic picture right here. Okay? Maybe it's the case that this is the key and I have no idea what it is accept that, it's as wide as these boxes are right here, okay? I'm going to specify the key by address. I'm going to specify the array by address. I’m going to tell me how many figures are in here.

I'm also going to tell myself how wide these individual boxes are, so I know how many times to 4-loop, and how far in memory to advance with each iteration, okay? I also have to be able to compare this pointer to that pointer somehow, or not the pointers themselves, but the material that’s at those pointers, okay, or at those addresses so that I can decide whether this matches this and I should return that value.

If on the next iteration it doesn't match, I have to be able to compare this value to that value. Or rather, the things that are at those addresses to see whether or not, there is a match. I have to rely on a comparison function to do that for me. Okay? So this is the prototype for the function I want to write (L search void * key) (void * base). I’ll call it base because the documentation for functions like this, actually calls it base. It just means the base of the array, okay?

I want to pass in (int, n). That's the number of elements that that the client knows is in the array to be searched. (int, OM sized) and that’s all the passing for the moment. I have to pass one more thing, but I'll do it a second, okay? I basically want to do the same thing up there, but I want to be able to be prepared to return an address as opposed to an integer, okay. And I have to implement this thing generically.

What I want to do is I want to be prepared to loop this many times. I don't think we're going to argue with that. Okay, with each iteration, what I have to do is I have to compute the (i) address or the address of the i's element. This is how you do this. This is actually something new. I want to set a void *. I'll call it elem address. That's going to be a variable that is bound to the tale of that or the tale of that or the tale of any one of these things, depending on how far I get, okay? Can I do this?

Your heart is in the right place if you try to do that. You are trying to jump forward i elements and you just want the compiler to know how big those things are, okay. It doesn't know how big they are. So you say, "Okay, well, I will tell it explicitly how much to scale each offset by," so at least numerically, that is correct, okay?

Take whatever the base address is and march forward this many quantum elements where the elements are just basically identified by size, okay? This is still point arithmetic, okay, and it's against a void *, so the compiler doesn't care or most compilers don't care that you know that this numerically this should work out.

I'm trying to manually synthesize the address of the (i) element but from an address standpoint it's like, "No, I don't care whether you are being smart over here. You are telling me to do point arithmetic against a type less pointer so I don't know how to interpret this and I'm not just going to assume that you are doing normal math here." So the trick is to do this. It's totally a hack, but it's a hack that is used everyday in generic C programming. I want to base and I want to cast it to be a char * and after I do that add i times the elem size, it is called the void * hack, at least in 107 circles it is. That's one full expression.

What I am saying is seduce this base or whatever number it evaluates to, to think that it's pointing to these 1-byte characters, okay? Then if I do point arithmetic against a char *, then pointer math and regular math are exactly the same thing. Does that make sense to people? So this right here would end up giving you the delta between the beginning of the array and the element that is of interest to you for that iteration. Does that make sense? Okay. This overall thing is a type char *, but when it is assigned to a void *, that's a fine direction t go in; it's going from more specific to less specific and it doesn't mind that. Okay? Make sense? Yes/No? Does not make sense.

You understand how this is the quantum number of bytes between the beginning of the array and the element of interest? You understand that that item applies to the base address of the entire array itself. The only reason I am doing this is to kind of get the compiler to work with me. It won't let you do anything like point arithmetic against a void *. You're even correcting, by scaling up by elem size, you are actually doing some of its work for you. Up here, this (i) right there is implicitly multiplied by size of int for you. Does that make sense? You have to nod or shake your head. It does not make sense.

Okay, this array is the base address, this is basically equivalent to * of array + i because this is a pointer and that's an integer constant. It multiplies this behind the scenes by size of int. Okay? It will not do that in a void * setting because it doesn't know what the implicit multiplication factor should be. So what I am doing here is I'm brute force doing the pointer math for the compiler, okay? I'm saying, "I'm dealing with void *s, I don't expect you to do very much for me on void *s so let me just cast it to be a char * so that I can do normal math against a pointer."

Some people cast these to be unsigned longs. I just happen to see char * more often than I see unsigned long, but they are both 4-byte figures where you can do normal math on them. It's incidentally normal math with char *s because characters are 1-byte wide, so the scaling factor is just 1, okay?

This is the number of bytes between the front of the array and the element that you are interested in. You assign it to elem address so that on this iteration, elem address, something like that or something like this can be passed as a second argument where this is the first argument to some comparison function. Okay, and it comes back with a yes or a no as to whether they match. Does that make sense?

Now if I write this this way, then the best I can do – if I don't pass in the comparison function, the best I can do is a generic memory comparison of the elem-sized bytes that reside at the two addresses to be compared. You could do this – this is one line. You could do this, that if it's the case that memcmp of key and elem address – elem size double = zero, you can go ahead and return elem address.

Now, the one thing you have not seen before, I'm assuming, is this memcmp function. It's like string comparison but it's not dealing with characters specifically. It compares this many bytes at that address to this many bytes at that address and if they are a dead match, it returns zero. It would otherwise return a positive number or a negative number depending on whether or not the first non-matching bytes differ in a negative direction or a positive direction but we're only interested in a yes or a no.

So that's why we do these double equals right here. Does that make sense? If you wanted to, I don't recommend it but you could do – if you hated double equals comparing integers, you could pass the address of your two integers here, passing size of int right there and it would do exactly the same thing that i double equals j does. Okay?

If you really just want to compare memory patterns and see if it's a dead match, then you can use this right here. This is going to work for Booleans, for shorts, for characters, for longs, for ints, for doubles and floats because everything resides directly in the primary rectangle. Okay? It will not work very well for character pointers or for C-strings. It will not work very well for structs that have pointers inside. Okay?

That point in the material that actually should be involved for the comparison. Does that make sense? So this is something that could work if you didn't want to deal with function pointers but you really should deal with function pointers. So let me just write this a second time. Then you go ahead and return null if things don't work out. Okay, you give it "n" opportunities to find a match and if it fails you just return zero as a sentinel, saying I couldn't find anything, okay?

Before I let you go, let me write the prototype for the function that we are going to write the beginning of Friday. (void * L search); the L is still for linear search. I want to pass in (void * key). I want to pass in (void * dates). I want to pass in (int n). I want to pass in (int OM size), and then I want to pass in the address of some function that is capable of comparing the elements that I know them to be. (int *) – I’m used to asterisks there. You don’t actually need them if you have a parenthesis, but I like the asterisk there to remind me that it’s a function pointer. And it takes two arguments. It takes a void * and another void * and that’s the entire prototype of that function.

Okay, now of course that is all supposed to be one line. That means of the fifth parameter to any call to L search absolutely has to be a function that takes two void *s, say void *s like this and this, okay? And somehow translates that to a zero, positive 1 or negative 1, okay? It has to basically have the same prototype that memcmp has, okay? When we write this next time, I am going to go through the implementation.

It really just more or less replaces memcmp with cmpfn right there and it does what we want. Okay, but it is interesting to see how you use it as a client and search an array of integers using the generic version. How you search an array of C-strings using the generic versions. It’s just very complicated to understand the first time you see it, okay? That’s what we’ll focus on on Friday.

[End of Audio]

Duration: 51 minutes