ProgrammingParadigms-Lecture03

Instructor (Jerry Cain):Hey everyone. I have one handout for you today. You don't have it in your hands because we're gonna pass it around. I just really have the one. I'm gonna have T.A.s pass it around during the lecture. I sent out an email on Saturday. Did anyone not get that email? Probably a handful. Oh, wow, nobody – everybody got it. That's great. Okay, you did not get it – the email? Okay. Well, if you can, like, email me directly after class so I can figure out what the deal is. The reason I say that is because I just realized over the weekend – this is totally my fault – that the section room for tomorrow, Skilling 183, seats 48 people. And I'm telling everybody to go to it. So I have some – I have to figure something out there. Normally that's going to be fine because most of you will just watch it on TV and get in the habit of doing that. But tomorrow I kinda want you to go. So I'm working on one of two solutions. I'm either gonna try and get a bigger room just for tomorrow for 4:15, or I'm gonna get a huge room at 3:15, still have the 4:15 section, and try and push everyone to go to the earlier one if they have any flexibility. So I will make a decision as to what – based on what rooms are available to me later today. And I will send out an email, which is why I'm asking about the email. Okay? So definitely stay tuned for a CS107 email after today. When I left you –what? Question right there.

Student: What was the time again?

Instructor (Jerry Cain):Tuesday at 4:15 are the normal sections. I may have one – may is the operative verb there – I may have one tomorrow at 3:15, just tomorrow, to accommodate the sheer number of people I expect to show up. Okay? When I left you last time, I was doing little asterisk and ampersand tricks. Let me do another one. I have a double D, and I set equal to 3.1416. And as a result, I actually get a fairly large figure in memory that's populated with pi. We'll decorate this with a D variable. And if I go ahead and do the following, CAR CH is equal to asterisk, CAR star, ampersand of D – there was a little bit of confusion about this. Because of that ampersand operation right there, the actual bit pattern that resides in the eight-byte figure that we're calling D, the bit pattern is actually irrelevant. This is an expression on the address of D. It doesn't have to look inside the eight boxes to figure out what's important here. It has to evaluate that address because that's there. It's seduced into thinking that it's actually storing the address – I'm sorry, that' it's an address of a single character. So when it's dereferences this right here, it goes and it embraces that single byte right there. Whatever bit pattern happened to reside there before is now pretending to be a character for the lifetime of this statement right here. Does that make sense to people? So if this happened to be – I don't know what it is – but suppose it's this bit pattern right there, okay? This variable called CH will get that very bit pattern. And if I go ahead and do cout << CH << ENDL, whatever this corresponds to gets printed to the console. Okay. That make sense to people? Okay, let me do one more little tricky thing here. If I declare a short, and I set it equal to 45, I get a two-byte figure that has a 45 in it. It's stored in binary, but I'm just gonna write 45 because it's easier to look at that. And I do this; this is where you get into a little bit of danger. Double star – I'm sorry, double D is equal to asterisk of double star ampersand of S. Most of what I've said prior applies here as well. This is one scenario where things are

a little bit mysterious. This right here evaluates to the ampersand of S. That's the address associated with that arrow, the number associated with that arrow. And this is a brute force reinterpretation of that address. So what's gonna happen is it's gonna say oh wow, that arrow now – it never pointed to a short. It actually points to an eight-byte double. So it will go, and not only include those two bytes right there, but the six bytes that follow it. Okay? Whatever bit pattern happens to reside there, provided there's no memory crash – and I'll explain why that could happen in a second. As long as it gets away with it, it's gonna go and embrace all eight bytes of those – eight bytes of information there, interpret it as an eight-byte double, and then assign it to this thing called D. So if this is a 45, followed by that as a byte pattern, then D would get 45 with this as a byte pattern. And when I print that out, it's gonna print out whatever number happens to be associated with that representation. Does that make sense to people? Okay. Now, I could do these for days, okay? And show you every little combination between asterisks and ampersands and double asterisks and whatnot. I want to move on and start talking about arrays and structs first because I think you'll just get more practice there. We'll start learning some more material. There was a question right here?

Student: Yeah. Are these examples gonna behave differently on low NDN?

Instructor (Jerry Cain):Certainly, yeah. The – well, as far as the bit copying is concerned, no. This ampersand, right here, is always the address of the lowest byte. But as far as how the – NDNS has to do more with interpretation and placement of bytes relevant to one another. As far as – there was these phrases in the handout that I kind of de-emphasized, but they're there. And since your asking, I'll talk about it. Little NDN. If I were to write down a two-byte short like that, and if I were to store the number one in that short, you would just say, "Oh, well the one just goes right here, and it's proceeded by 15 zeros." Does that make sense to people? Okay. That is true on about half of the systems in existence at the moment. This right here – not only is a representation for positive one as a two-byte short – it happens to be stored in what's called big NDN format. And the best way to remember that – it's kind of arbitrary as to what big versus little means in this context. I just remember it as the lowest byte stores the bits that correspond to the largest contributions of magnitude. Does that make sense? Okay. And since on is such a small number, that's why you have all these ones over here. On some machines, in particular the Linux machines that you're probably working on, it would store this with the bytes in the reverse order. Okay? It would actually have the one right there, proceeded by seven zeros, followed by eight zeros right there. And it's just when it goes it interprets it as two-bye short. It actually assumes that these are bits zero through seven. And these are bits eight through fifteen. Does that make sense? So if you actually were to – and you could do this if you wanted to – if you were to copy a two-byte short from a Linux machine to Solaris machine, and just do it on a byte copy level, you wouldn't get the same numbers on different machines. Okay, you would get one on a bit NDN machine here. You get 256 – I'm sorry – yeah 256 on little NDN machine. Does that make sense to people?

Okay. For the most part you don't have to worry about NDNS at all. There's one aspect in assignment two that happens to deal with it, but I'm more or less insulate you from it.

Okay? Just something to be sensitive to. What I want to talk about now are structs, how they work. How arrays work. How arrays of structs, structs with arrays inside all work. Given what we know already – let's kill this. Let me go ahead and declare this as a struct right here. Very simple. Struct fraction int num int denom. That right there, that's the C way – I'll assume we're in C++ for the moment. That's enough to declare fractions stand alone as a new type. If I do this, fraction – let's say pi as a variable name. As a result of that, I obviously get enough memory to store a fraction. In 106B and 106X, and maybe 106A as well, you drew them as the somewhat loose rectangles around two boxes. Okay, I want to be a little bit more structured than that. I want to recognize that the amount of memory that's set aside for the struct fraction, not surprisingly, is eight bytes. Okay? It's basically the sum of some of its parts, and it actually packs all of those bytes as tightly as possible. I'm gonna draw this as eight bytes. I'm gonna emphasize the fact that it's really four byte stacks on top of four more bytes. The address of the entire struct is always coincident with the address of the first field. So looking at this, and assuming that that's a picture of one of these things, you know that this is the num field. In this case, it'll be pi dot num because that's the way I declared it right there. And stacked on top of that, four bytes above the base address of the entire thing, would bit pi dot denom. Okay? So when I draw that arrow right there, unless they give you some context, you don't know whether it's an int star pointing to the address of the num field – I'm sorry, storing the address of the num field or the address of the entire struct. Okay? When I do this, not surprisingly, that places a 22 in the lower four bytes of the entire figure. The more technically accurate way of saying it is that it actually stores a 22 to the field that's at an offset of zero from the base address of the entire struct. Okay? That's why 22 gets placed there. When I do this and store a seven some where – because it recognizes base on this definition, which it certainly sees before it sees this line right here, it knows that denom is stacked on top of num because num is a four-byte integer, but denom is four bytes above the base address of the entire thing. And that's how it knows where to put the 29 zeros followed by three ones for the seven. Okay? If I go ahead and do this – this is where things get crazy – if I go ahead and do this, ampersand of pi dot denom. I'll do that. You don't technically need it, but that makes it clear what you're taking the address of. I have an int star, right, unless I do this. Okay? And now I have the address of a fraction. So what happens is just on the fly, it stops thinking about this address right there as a stand-alone integer, or pointing to a stand-alone integer. Now it has this picture, all of a sudden, at the moment that it's addressing this eight-byte picture that overlays that space right there. So when I go ahead – let me actually draw this a little bit more accurately. I go ahead and do fraction, asterisk, and I do this, num is equal to 12. The arrow comes after something that at that moment is assumed to be the address of an entire fraction struct. So the arrow travels to that struct. There's not much traveling to do because it's already there. And then it goes inside and identifies the num field as the place that should receive the 12. Where is that num field? It is right here. Does that make sense to people? Okay. So if I go ahead and I just print out cout << pi dot denom, behind pi dot denom's back, it was changed from a seven to a twelve. Okay? If I do the same exact thing, fraction, star, ampersand of pi dot denom, arrow denom is equal to 33, it's going to – it's not going to be concerned about the fact that I really don't own the space above what is truly pi dot denom. The way the mechanics work takes the base address of this. Oh look, it's a fraction star now. Go four bytes beyond that to find out where the 33 belongs. It's gonna smear down the four-byte

representation of 33 in this space right here. And there's no legal way to get to it and print it out, but if I did this again to the right of a cout statement, it would print out a 33. Does that make sense to people? Okay. Good. Let me do one more thing. Actually, let's not. Let's go on to arrays. Int array. Very different from Java. We didn't talk about arrays in CS106B and 106X as much as we will in 107. Question in the back?

Student: Yeah. Sir, I don't know if I caught you correctly, but did you say that when you set the fraction and then ampersand, p dot denom, the num to 12, does that mean when you access pi dot num – or sorry, pi dot denom that it's gonna be 12 and [inaudible].

Instructor (Jerry Cain):That is correct. Right. You happen to invade pi dot denom's space using some quirky syntax. Okay? Just because you happen to know that pi cot denom resides above pi dot num, just because you reinterpret the address to be associated with a different data type, if you happen to operate on space that overlays the original pi dot denom, then you're affecting what really is pi dot denom's value.

Student: So is there to access the denominator – or the denom of the four-byte representation within the four-byte representation?

Instructor (Jerry Cain):Within the –

Student: How – basically, how can you access that 33 without doing another [inaudible].

Instructor (Jerry Cain):You actually can't. I'm sorry, you certainly could use this expression again to do so.

Student: You can't access it any other way?

Instructor (Jerry Cain):If you wanted to – I mean this will be more clear after I do the array example, but if I wanted to – since you're asking – pi dot denom, fraction star. Okay? Do you understand that that's the address of the top two thirds of the drawing? What I could do is I could do something like this. That's a little sneaky. That'll be clear after I do the formal array example that I'm covering up right now. But that's effectively a dereference to go and get to the denom field. Okay? I wouldn't even have to do this necessarily. I could just do address of pi of one dot num is equal to, or print that out or something like that. Okay? I mean this will become more clear after I talk about the array business a little bit.

Student: Because what's at the one index is actually a fraction?

Instructor (Jerry Cain):It literally is eight bytes beyond what's at the zero index. Okay? You don't get that; don't worry. I'll start with a simpler example right here. This right here, you know this already. It allocates 40 bytes of memory, okay, for the ten bytes that are being set aside and under the jurisdiction of this key word called "array." So I'm going to draw it this way, and do a module of five – spit it out. You know that this is a zero index, and you assigned to it using an array of zero. This is an array of nine. So

when I go ahead and do something like this, array of zero is equal to 44. I put a 44 there. You know this. If I do array of nine is equal to 100, and 100 goes there. What you may not recognize is that array itself is synonymous with the address of the zeroth entry. Okay? Let me write that down as, like, a little – like a little theorem. In the context of that declaration right there, array is completely synonymous with ampersand of array of zero. Okay. That's why when you pass an array to a helper function, or any function whatsoever, you're not passing the entire array, you're just identifying the location of the zeroth entry, and from that you can access anything legitimately beyond it, as long as you know how long the array is. Okay? If I go ahead and do this, let's create some tension.

45, and obviously zero, one, two, three, four, five, that's nothing new. If I go ahead and mess up, and I don't understand four loops, and I don't understand arrays well enough to not make this mistake yet. If I go ahead and I write down the number one, it's consistent with the offsetting that's done relative to the base address of the entire thing. This right here assigns a 44 to the int that is at zero ints forwards of the base address. Go ahead nine quantums of integers to find out where the one hundredth should go. Go ahead five. Java's a different story, but in C and C++, there's no bounce checking done at all on raw arrays, and that's exactly what this thing is right here. So, when I do this, it really says, oh. Well, that's interested in that address right there. This is ten is interpreted to be ten times the size of an integer, which is four, for a 40-byte offset from the base address right here. So it goes right there, and it leaps forward forty bytes to the base address of what it has no choice but to assume is an integer space. So it's going to go down. Whether it's going to cause problems or not is a different story. It'll try and place a one right there. Okay? If I do this, f 25 is equal to 25, and somewhere over here, a 25 is laid down. Okay? It actually even tolerates negative numbers. It's that brute force – I don't want to put zero. That's not very useful, 77. It would march back one, two, three, four places to figure out where to place this 77, and that's how memory as a side effect would be updated by these bogus little statements right there. Okay? Does that make sense? Question in the back?

Student:[Inaudible] make the assignment of the right 10 if it's going to do that anyway?

Instructor (Jerry Cain):I'm not sure what you mean. Say it again.

Student: Say an array 10, like, you initialize it 10 by spaces, but like, what's the point of initializing it if it's just going to do what's – basically do what you want when you get inside of it.

Instructor (Jerry Cain):That is true, actually. This right here is really just documentation for how much space is being allocated. And then you're supposed to write code – I'm not saying this is good code. I'm just saying its code. Okay? You're supposed to write code that's consistent with the amount of space that you legally have. But this, this, and this just work because there's no bounce checking. It doesn't look arbitrarily far backwards to figure out whether or not it's an in-range index. So when it gets away with this, and it compiles, and it runs, it's just gonna put a one where it assumes that the eleventh entry would be, or the twenty-sixth entry, or the negative fourth entry. Okay? Does that make sense to people? Does that make sense? Okay. Yep?

Student:[Inaudible] the memory?

Instructor (Jerry Cain):It doesn't in C and C++, not at all. All it does is instruction for that one declaration as to how much – how many variables, more or less to – I'm sorry, how many ints to set aside space for. But once you do that, like, there's no – the length of the array is that. But the length of the memory figure, it's not exposed to you. So there's no way to recover it. That's why you always pass around the length, width, a raw array in C and C++. Okay? You use vectors more than you did raw arrays in C in 106B, but we're gonna be more C programmers than C++ programmers for the next few weeks, so we don't have vectors because we don't have classes. And we don't have templates. So we actually have to take this approach right here. Okay? Yep?

Student: So you use the address of pi and the X sets it as an array there. Is it gonna know that the sides of each element is a fraction?

Instructor (Jerry Cain):Yep. That's – it uses the data typing of whatever pi is right there, and because ampersand of pi is an int star, it knows that if you automatically – if you just all of a sudden start treating it as the base address of an array, even if it is really only an array of length one, it's gonna deal with the default offset of eight because that's how many bytes are in a fraction. Okay? So this will become a little bit more clear after I put a few more little theorems over there. Okay? When you do this right here – let's say it this way. Array of K, where K is an arbitrary integer, it is – the address of that thing is completely synonymous -- and you did not see this all that much, if at all in 106. It is synonymous with this right here. Okay. So the first line isn't array so much as it is array plus zero on the left hand side. Okay? This right here, given that example, array is of type int star. There's no storage for array. It's not like the address, the base address of the array is stored anywhere that you can manipulate. But this is of type int star. If this is assumed to be an integer, which it is in this example, then you're not doing normal arithmetic here. You're doing what's called pointer arithmetic. And it knows that you're not going to be dealing with arbitrary bytes inside an array. You're only supposed to be concerned with the boundaries that separate the space where one int ends and another one begins. So whenever this is understood to be a pointer right here, this number isn't added verbatim. It is automatically scaled by the size of the figure being addressed. And it knows what the figure is based on the type system. In this case, it knows that it's pointed to an int. That's why there's – those are four bytes. That's why all these rectangles are seemingly four bytes wide. In this example up here, ampersand of pi evaluates to fraction star. So when I start treating it like it's an array even though it's not, I have no choice but to rely on this rule right here to figure out where the oneth, counting from zero, fraction would be, starting at this address. Okay? And that's why it advanced eight bytes beyond the base of that entire drawing to figure out where to start dealing with things. Okay? Does that sit well with everybody? Yes? Yep?

Student: Is there any way to get access [inaudible].

Instructor (Jerry Cain):There is. You can actually use some casting tricks. I'll do that in a second, okay? I will do that, like, probably in two or three minutes, but I'll do a really

good example for that question. Okay? What I hope is a good example. I should say it that way.

Student: Permission for the array [inaudible].

Instructor (Jerry Cain):That's correct.

Student:[Inaudible].

Instructor (Jerry Cain):That is correct. So just because I write a one here and I have code that actually tries to do it, doesn't mean that while it's running, it's gonna succeed. If it succeeds, it does place the bit pattern for one there. It might also crash. Okay? Or it might actually succeed. But this space right here? We'll see this very shortly. This space right here and this space right here is gonna be associated with other local variables that happen to be declared above this and below this. There's no impact here because this is the only declaration. But if I were to declare int I right there, and double D right there, the model we're going to use – and this is the model that's really used – is gonna packs all local variables into a little thing called an activation record. That's just fancy terminology for the block of memory that's set aside for all local variables in a function. So if you touch this right here, you're really touching some other local variable. You're touching the one over here that was declared after the array. This is the way it kind of works out. Okay? Does that make sense? Okay. There's a couple more rules I want to talk about here. When you dereference this right here, if you put an asterisk in front of this ampersand, they kind of negate one another. So this is synonymous with array of zero. The extension of that for this line is that if I put an asterisk in front of this, the pointer arithmetic is done first so it computes the address of the integer you're interested in, and then the asterisk actually brings you into that rectangle. It is synonymous with that right there. So that's why when you do something like array of negative four, you're really doing this. Oops. Pointer arithmetic brings you not four bytes before, but 16 bytes before that address.

You dereference it to actually sit in, and find yourself in a rectangle that's capable of receiving the 77. Does that make sense to everybody? Okay. That's great. So in a second I will start mixing arrays and structs, but to get to your point with regards to how do you access the internals if you want to do it. You rarely want to do it, although there are – actually it turns out that there are features of assignment two that rely on this type of knowledge. I'm not encouraging you to write this code, but I don't see the disadvantage of understanding it. If I go ahead and declare, let's say, an int array – I'll keep it small – five. Oops. I get this right here, one, two three, four. If I go ahead and set array of three equal to – let me do 128. Uninitialized, left uninitialized, left uninitialized, left uninitialized. I actually put a 128 there, and I'm drawing in the right half of the box because that's really where the bits will be updated. Everything to the left of the 128, right here, will be all zeros. And this will be one followed by seven zeros. I'm just emphasizing the fact that the 128 happens to fit in the lower of the two bytes. Okay? If I do this, the data type of that is int star, right, unless I do this. Okay? Now, ARR is brainwashed momentarily into thinking that it addresses a short. And there, incidentally, is space for ten shorts there.

Okay? The way ARR, or the way the result of that expression sees it, that's short of zero, short of one, short of two, short of three, short of four, short of five, short of six. Make sense? Okay? This is kind of what you were getting at, I'm assuming. Zero, one, zero, two, four, six. It's gonna write a two in that byte right there. Okay? So when I go ahead and I cout << ARR of three << ENDL, you are not printing out a 128. You're actually printing out 512 plus 128. Everyone know where the – where I'm recovering that 512 value from? Okay. If this is the number two, and I multiply it by two eight times to get into that position right there, that's really two to the ninth plus two to the seventh. Okay? And so it's going to print out whatever that number is right here. Okay? I can go arbitrarily nuts with all of this casting. If I wanted to set ARR of one address, and I want to cast that to be a CAR star, I want to add eight to that, and I want to cast that to be a short star. And I want to find the third short after that and set it equal to 100. I think I have the patience to go through with this and show you what's going on right here. ARR of one is that box right there, so the ampersand is that right there. Pretend just for me that you're a CAR star so I can do something funky and add eight to you, but have it mean eight time the size of CAR plus two plus four plus six plus eight. So that's the address of this right here. Okay? You're a CAR star. No, you're not. You're a short star. Okay? Pretend you're the base address of an array. I don't care how long of an array it is. Just go three shorts forward of that short star that's right there to figure out where to write a 100. This is the zeroth one. This is the oneth one, the twoth one. The third one. This is where the 100 would go. Okay? Don't write code like this; just understand it. Okay? This make sense to people now? Okay. Let me start blending structs and fractions to get more interesting examples. We're – come Wednesday, we're gonna be able to do meaningful stuff with this knowledge. Right now it's all gibberish, and it seems like it's just contrived code. It is certainly contrived code because the examples need to be small and focused.

But once we understand how to deal with memory – and that's what we're really doing with all of these examples – you'll be able to take the understanding of memory and write meaningful generic code in C. C we don't have templates. That's how we dealt with generics in C++. In C we have to leverage off – over the fact that we know the size of everything, and we know that bit patterns represent vales to be able to write a generic binary search, or a generic linear search, or a generic swap function, or a generic vector, or things like that. Okay? And that's what Wednesday and Friday, and probably next Monday are going to be all about. Did you get this example? Okay. You'll – if you don't get it yet, section handout come next Tuesday, not tomorrow, will deal with more of this stuff. Okay? Let me go on to structs with arrays in side of them. I'm gonna need two boards. That's why I'm erasing so much. I should just erase with the chalk. Here's the struct definition I want to deal with. Struct student. Okay? I have a field inside. I want to store an exposed character pointer. You're not used to doing this because you had a string class in C++. We actually don't have those in pure C. They're always represented as character arrays, okay, where the characters in the string are laid out side by side. Rather than there being a period at the end, there's what's called a "null character," The backside zero character that's at the end. This one's gonna happen to reside as a string outside the struct. All I want to do is I want to store the address of the zero character of the entire name. That's different from this, SUID of eight. I want to store the individual digits of a seven-digit SUID in an array that's wedged inside the struct. This will become clear from

a picture in a second. And then at the bottom I just want a normal integer num units. And there we have our definition. Okay? What's a picture of one these things look like? It looks like this right here. There's my CAR star. There's my static character array of length eight. And there's my num units field. So this is a sixteen-byte struct, okay? You're not used to looking at these things this way, but in memory diagrams, at least usually – at least for the next day, you read left to right, bottom to top because you're always worried about the lower addresses. The address of the entire struct is coincident with the address of that CAR star. Okay? So to see this arrow, you don't actually know whether or not it's a student star or a CAR star star. Yes, we'll be dealing with double pointers. Okay? If I go ahead and declare four of these things in an array, student, pupils of four, then I get four of those things laid out side-by-side. The way of laying down the elements, the base elements of an array, is the same whether you're dealing with Booleans or ints or doubles or structs. So actually you're gonna have four of these things. Draw those right there to make it clear that we're – that's the zeroth, the oneth, the twoth, the third. Okay. So I have all 64 bytes of memory for my packed array of four items. Each item is a struct, and the same skeleton, or the same view of memory, overlays each of the four quantum elements.

So when I do this, pupils of zero dot num units equals 21, you know that 21 goes somewhere, and this isn't too bad. You know it's gonna go in the space that's dedicated to the num units field of the very first student struct. Okay? If I do this, pupils of two dot, let's say, name is equal to – there's this function I want to talk about – strdup Atom. S-T-R-D-U-P, strdup is actually shorthand for string duplicate. Okay? So what this does as a function is it dynamically allocates just enough space to store the string – in this case Atom – and then it actually writes down Atom in that space, and as a function returns the address of the capital A. Okay? These four things right here are all local variables – I'm sorry, the entire array is a local variable. It resides in a part of memory called the stack. I'm assuming you've heard of the word "stack" before. You may not have heard it talked about in the case of memory. But the dynamically allocated string – and this is dynamically allocated – that is drawn from a part of memory called the "heap." Logically, we assume that it's five bytes. It actually makes space for that backslash zero. Okay? And then the address of that new figure right there, after it's been initialized with whatever this string logically is, gets returned, and it's dropped in the name field of the third, counting from zero, okay, struct. So this gets placed right there. Okay? That's very different than this type of setup, where pupils of three dot name is equal to pupils of zero dot SUID plus, let's say, plus six. There's a lot going on in that line. Let's just look at the right hand side, pupils of zero dot SUID. Pupils of zero SUID of zero is that right there. Pupils SUID of four is right there, but I don't have any array index dereference going on there. I have just the raw array name right there. That's synonymous with that arrow right there. In spite of the fact that this is a big, nasty expression that evaluates to a pointer, when I add six to it, it's doing pointer arithmetic against a CAR star. Okay? So the six is effectively, even though it doesn't matter, it's scaled by the size of a character, which is one. And so the overall right-hand side expression is the address of that character right there. Does that make sense to people? Okay. The tale of that arrow is assigned right there. All I did was I assigned an actual value to the name field of the very last student in that record. It happens to be the address of something that resides inside the entire figure. Okay? If I do this, pupils of one – oops, messed up. Str – not dup – cpy of pupils of one

dot SUID four zero four one five XX. Right there. Strcpy is like strdup, except it doesn't actually allocate any memory. It assumes the address where you should copy the string is identified by the first arguments. So what this does is beneath the surface – in strdup as well, but specifically in strcpy – there is some little four loop that keeps on copying characters one after another until it finds a backslash zero, and it copies that as well. In fact, strdup, after it calls C is equivalent of operator new, which is called malick; it actually calls strcpy. This right here, on this one-by-one basis would write a four right there. And then a zero, and then a four, and then a one, a five, an X, an X, and a backslash zero would be written right there, and then it would return. And it's completely useful because of its side effect of copying characters around. You have to make sure that the address you pass in there actually points to character space that's really under your jurisdiction because it's gonna try and write characters to whatever address that's specified there. You better make sure it's a good address. Okay? This one right here, strcpy again, of pupils of three dot name, one, two, three, four, five, six. – and that's enough; don't worry about the fact that that's not really a Stanford ID. It was 70 years ago, I'm sure. Pupils of three dot name. That evaluates to whatever this evaluates to. That means that location right there, okay, is what's identified as the place where characters should be written. Okay? It follows exactly the same recipe that the first called of strcpy did. It's not as if this byte and this byte are auto-declarated behind the scenes as things that can always store characters. And it's not as if that's turned off right here. As far as strcpy is concerned, it sees this as a base address of an arbitrarily long character sequence space where characters can be written. And so what's going to happen is it's going to write the digit character one right there. It's going to write the digit character two right there.

It's gonna do exactly the same thing right there with a three, a four, a five, and a six. It's gonna write a backslash zero – oops, not there – in the left-most byte of that name field right there. Okay? Does that make sense? And then strcpy's, like, I did my job. I'm awesome. I'm gonna return back to the main function. So when you come back, if you want to print out the number of units this student was taking, it's a lot. Okay? It is three times to the 24th plus four times two to the 16th plus five times two to the eighth plus 6. They'd have to petition to do that. Okay? If I go ahead and I print out this string right here, and I actually pass this in, if I do cout << pupils of three dot name, it actually would print one, two, three, four, five, six, and that's it. Okay? That's because it just receives the address of something it trusts to be a character followed by probably another one followed by yet another one. It just crawls over consecutive bytes of memory until it incidentally finds one with a zero in it. Does that make sense to everybody? Okay. Again, you will not be writing code like this, okay, but you should be able to understand, at least believe that the drawing I'm putting here is consistent with the code. You had a question?

Student: Yeah. [Inaudible].

Instructor (Jerry Cain):Yeah, that pupil's three dot name. So this right here, the name field – it's not ampersand of name. So I don't pass the address of this box. I have pupil of three dot name evaluate itself. Okay? So if this is the number 1,000 in here, it's because the address of that box right there is really a 1,000. And that's what passed as strcpy. So it

starts copying characters to address 1,000, and then 1,001, 1,002, etcetera. Does that make sense? Okay? Question in the back? No. Okay, you guys are good. One of the thing I – yep, right there.

Student: Look at variables in the string. What happens [inaudible]?

Instructor (Jerry Cain):If I – this right here, before this block ends, unless I want to pretend that atom is a helium balloon, and I want it to fly off and never be recovered, I would have to free it before this code block. And I could just do that by passing pupils of two dot name to free. Okay, free is the C equivalent of this delete thing you're familiar with. Okay?

Student: It doesn't really tell us [inaudible]? Teacher:

Nope, not at all. That's Java. That's not C++. Okay. One of the line pupils of 7 dot SUID of 12 – let's not do that; let's do 11 – is equal to the character A. Just because there are structs involved doesn't mean that it intimidates the executable. It will go ahead and it will do the manual pointer arithmetic to find out where the seventh student would reside if this address, if the array actually existed there. So I would go to not the zeroth, the oneth, second, or third. I better go to the fourth, the fifth, the sixth, the seventh. There's a gesture, a little phantom halo, around the space that we're identifying, or pretending, these pupils of seven. Then I jump to its SUID field. That would reside and begin right here. As if I legitimately had space for eight characters right there. It even double whammies the system and goes beyond that array boundary. This is four – I'm sorry, this is zero. This is four. This is eight, nine, ten, eleven. It would write this scattered A, 65, in that one little byte over there in memory. Would it succeed while it's running? If it crashes, no. If it doesn't, yes. Okay? That's just the way it will work out. Okay? Does that make sense to people? If I were to go ahead, and I were to print in this state right here, if I were to print just the address itself, all I know is that the other three bytes are uninitialized. If I print this entire number, okay, all I can tell is that it would be less than two to the 24th. That's all I know because I zeroed out the really large contribution to the overall thing. Does that make sense? Okay. If I were to print out this right there, if I were to pass that CAR star, it has no idea that it's the address of a character that happens to be in larger string that starts before it, so if I were to pass that address to cout <

What happens is that you've probably declared two integers, X seven, int Very good. Okay. You guys are doing okay? Good. What I wanna do now, is I wanna start talking about how to write generics in C. We have enough experience with this memory business so that I can actually write a real function in C that leverages off of this stuff. Let me just write a function I know you've seen before, and it's actually charmingly simple for us to go out because this is all very difficult compared to what I'm about to write. Actually, this board's better. I want to just write a really simple function, and use advanced memory terminology to describe what happens. Void swap int star – actually, you probably haven't seen this version before if you've used references in the past. What happens is that you've probably declared two integers, X seven; int Y is equal to 117. And I'm concerned with the call to swap, where I pass in the locations of my X and Y variables. C – and I'm

writing up here. C function right here has no templates. That's relevant. It also has no references. Okay? So there are few meanings – fewer meanings of the ampersand symbol in C. What I'm doing here is I'm assuming I own X and Y as little jewel boxes, and I pass the addresses of those to the sway function so it knows, at least, where to go to move byte patterns around. That's effectively what's done by the swap when you think about it memory terms. Okay? Does that make sense? So this is a function I haven't written yet, but I know that this thing called AP and BP – the P is there for just to remind myself that it's a pointer – this points to the X box and the Y box that has a 117 in it. So what I want to do is I want to exchange the one – I'm sorry, the seven in the 17. The way I do this is I declare a tenth variable, and set it equal to what I get by traveling from the AP pointer to the space it addresses. So I get temp right there. How is it initialized? It's not set to this number. The asterisk says please hop forward once to find the place that should be copied. The bit pattern for that seven is replicated right there. Because tenth and the space addressed by AP are both ints, the bit patterns mean the same thing in both contexts. Then I do this. A little bit more involved, but you understand, certainly, what's going to happen. You may not – if I wrote a more difficult version of this type of function, it might not get it. But what happens here is the space addressed by AP – not this space right here, but the space addressed by it – is identified as the L value, or the recipient of whatever the right-hand side evaluates to. The right-hand side evaluates not to BP, but to what it addresses. Okay?

So this 117 is replicated right there. The four-byte representation of 117 is replicated in the space addressed by AP, and then finally I do this. BP addresses whatever was stored here previously. And that's how I get a seven right there. Let's get a better seven. Okay? Now, what I did there, algorithmically, had very little to do with ints. The only part of the fact that – the only fact about ints that was involved was that the figure's being rotated and swapped for four bytes. Okay? Make X and Y floats. Make this float star and float star, and make that a float. The pictures can even stay the same in terms of the drawings – in terms of the sizes. They're still four bytes, and as long as I exchange all these things, okay, then I'm going to effectively achieve the swap, even though I don't necessarily care that they were floats versus integers. Okay? If I pass in double stars, or CAR stars, or bowl stars, or struct student stars, the same rules apply. Okay? It's bit pattern swapping is what it's – what it really is. Okay? You know enough about generics from CS106 PM, 106X to know that we would probably use references – because references are prettier, right – from this point forward. And we would also templatize it if we wanted the same block of code that we write to be used in different type scenarios. We have neither one of those in pure C. But there are several situations where you do benefit by actually going the extra mile and making the code you write generic. Okay? Well, it's not pretty. Turns out it's actually kind of – it's something of a hack to write a generic function in C, but it is the way it's done. And once you understand memory really well, you stop thinking of it as a hack, and you start to see it as very, very beautiful. Okay? As the way it actually works – because you understand what's happening on your behalf when you swap these two figures – and you just specify the addresses. Or you linear search this array, and the algorithm for linear search is the same whether or not ints or strings or struct students are involved. Binary search the same way. Merge sort, quick sort, all those things you learned about, and templatize, in 106B still can be done in languages older than C++

using this information about memory that we've learned over the last two lectures. Okay? So come Wednesday, I will go generic on you with this function right here. And frame it in terms of generic pointers and generic byte swappers. Okay? Have a good night.

[End of Audio]

Duration: 53 minutes