ProgrammingParadigms-Lecture02

Instructor (Jerry Cain):Hi, everyone. Welcome. I have four super handouts for you today. If you haven't gotten them yet, feel free to just sit down. We're gonna probably make it a point because there's so many people in the class to just hand them out while I start lecturing. This way we don't have this big bottleneck of people trying to get in by 11:00. The four handouts are posted to the web page. The mailing lists were created last night. And I just looked at it this morning, and there were 245 email addresses on it. So it looks like it's working. I haven't sent anything to the email list yet, but I will just contrive a message later this afternoon, and send it to everybody. And if you don't get that by Monday morning, when I make an announcement saying, "If you didn't get that message let me know," then I'll investigate as to why you're not on it. SEPD students, I'm not sure that you're actually on the mailing list yet. That system runs a little bit differently, and usually they push your email address onto the mailing list a little bit later. I'm not sure why, but – so if you don't get an email over the course of the weekend, then just let me know. And I'll see what I can do to fix it. I'll also post announcements to the web page so that you at least can get them. What I want to do is I want to start talking about the low-level memory mechanics, so that you understand how data – things as simple as Booleans and integers and floating-point numbers and strucks and classes – are all represented in memory. It's very interesting, I think, to understand how everything ultimately gets represented as a collection of zeros and ones. And how it's faithfully interpreted every single time to be that capital A, or that number seven, or Pi, or some struck that represents a fraction, things like that. And we'll just become much, much better C and C++ programmers as a result of just understanding things at this low of a level.

So, for the moment, C and C++ are the same language to me. So let's just talk about this. Let me just put a little grid up here of all the data types that you've probably dealt with. You've probably dealt with boole. You've probably dealt with CAR. I'm sure you have. You may not have dealt with the short, but I'll put it up there anyway. You've certainly dealt with the int. You've certainly – well, maybe not dealt with the long, but let's just pretend you have. You've probably seen a floats. You've probably seen doubles. And that'll be enough for me, enough fodder for the first ten minutes here. These three things – I'm sorry. These three things right there are certainly related. They're all intended to represent scalar numbers. Obviously, this represents a true or a false. This represents in our world one of 256 characters. We usually only pay attention to about 75 of them, but nonetheless, there are 256 characters that could be represented here. These are all numeric. These take a stab at trying to represent arbitrarily precise numbers, okay? The character is usually one byte in memory. At least it is in all C and C++ programmers – program compilers that I know of. This is typically two bytes. The int can actually be anywhere between two and four bytes, but we're going to pretend in this class that it’s always four bytes, okay? The long, for the time being, is four bytes of memory. There is another data type, which isn't really common enough to deserve to be put on the blackboard, called the long long, which gives you eight bytes of memory to represent really, really large decimal numbers. They'll come up later on, but I'll talk about them if I ever need to. The float is four bytes. It somehow tries to take four bytes of memory and represent an arbitrarily precise number to the degree that it can, given that it's using a

finite amount of memory to represent a number that requires and infinite amount of precision, sometimes. And a double, I've seen them ten and twelve bytes on some systems, but we're just gonna assume that they're eight bytes. Now, that's the most boring spreadsheet you could possibly introduce a class with, but my motivation is that I want to uncover what the byte is all about, and how four bytes can represent a large frame of numbers, how eight bytes can represent a very large set of numbers, and actually do a pretty good job at representing numbers precisely enough for our purposes.

So forget about bytes for the moment. Now, I'll go back to the boole in a second, because it's kind of out of character as to how much memory it takes. But I'm interested, at the moment, in what is less commonly called the binary digit, but you've heard it called the bit. And Double E students and those who enjoy electronics think of the binary digit in terms of transistors and voltages, high and low voltages. Computer scientists don't need to view it that way. They just need to recognize that a bit is a very small unit of memory that can distinguish between two different values. Double Es would say high-voltage, low-voltage. We don't. We actually just assume that a single bit can store a zero or a one. Technically, a Boolean could just be mapped to a single bit in memory. It turns out it's not practical to do that. But if you really wanted to use a single bit to represent a Boolean variable, you could engineer your compiler to do that, okay? Bits are more interesting when they're taken in groups. If I put down eight bits here – I'm not even going to commit to a zero or a one, but I'm gonna draw this. This isn't zero over one as a fraction, this is me drawing eight bits – let me just draw one over here so I have some room – and put a little box around each one of them in this binary search way, okay? And I have this big exploded picture of what we'll draw several times to represent a single byte of memory. Now, the most interesting thing to take away from this drawing is that this little box right here can adopt one of two values. Independently of whatever value this box chooses to adopt, etc. In fact, there are eight independent choices for each of the bits. I'm assuming that makes sense to everybody, okay? That means that this, as a grouping – a byte of memory with its eight bits that can independently take on zeros and ones can distinguish between two to the eighth, or 256 different values. Okay, and that's why the Ascii table is as big as it is, okay? 65 through 65 plus 25 represents the alphabet. I forget where lowercase A starts. But every single character that's ever printed to the screen or printed to a file is backed by some number. I know you know that. When you look in memory to see how the capital A is represented, you would actually see a one right there – I'm sorry, I forget where it is actually – a one right there and a one right there. I'll draw it out and explain why that's the case. Because capital A is backed by the number 65, we don't put things down in decimal in memory. We put them down in base two. Okay? Because that's what – that's the easiest thing to represent in a bit-oriented system, okay? That make sense to people? Okay. So if I say that the capital A is equal to 65, you have to stop thinking about it as 65. You have to think it about it as some sum of perfect powers of two. So it isn't 64 – it isn't 65 rather, it's actually 64+1. One is two to the zero. A two is two to the first. There's none of that. Four is two to the second. Eight is two to the third. Sixteen is four. Thirty-two is five. Sixty-four is six. This is actually two to the sixth plus two to the zeroth. Make sense? Okay. As far as the representation in a box like this, if you went down and actually examined all the transistors, okay? The eight that are laid side-by-side in a single character, but byte of memory, it would look like this.

And in order to recover the actual decimal equivalent, you really do do – you really do the power series expansion, where you say there's a contribution of two to the sixth because it's in the sixth – counting from zero from the right, the sixth position from the end of the byte. This contributes to the zero, if you can look at it as having contributions of two to the first, and two to the third, and two to the seventh that are weighted by a zero as opposed to a one, okay? That make sense to people? Okay. So that's good enough for characters. Let's graduate to shorts. Some people are very clever when they use shorts. A lot of times they'll – if you know that you're going to store a lot of numbers, and they're all going to be small, they'll go with an array of shorts, or a vector of shorts, knowing that there really will be some memory savings later on. The short, being two bytes, just means that two neighboring bytes in memory would be laid down. Those are the two bytes at the moment – would be laid down, and the two to the sixteenth different patterns that are available to be placed in that space. It can distinguish between two the sixteenth different values. That make sense to people? Okay. So I'll just make this up. I'll put lots of zeros over here, except I'll put one right there. Did I put too many? Yes. I did. And this is a wide bit. Okay. So as far as the number that that represents – I should emphasize that technically, you can map that pattern to any number you want to, as long as you do it consistently. But you want to make the computer hardware easy to interpret. This place right here means that there's a contribution of two to the zeroth, or one. There's a contribution of two to the first, contribution of two to the second. So there's a two and a four that are being added together. Two to the zeroth, two to the seventh, two to the eighth, two to the ninth. Okay, so there actually is a contribution of two to the ninth, which is 512. So this really is the number that's represented by this thing. It would be 512, 516, 518, 519 would have that bit pattern down there, okay? Does that make sense to people? If I have another one – oops, I don't want that there. I have one zero, followed by all ones and all ones, okay? I know that if this had been a one right there, then that would have been a contribution of two to the fifteenth. Does that make sense to people? Okay. Zero followed by all ones in binary is like zero being followed by all nines, in some sense, in decimal. It's one less than some perfect number that has a lot of zeros at the end, okay? Does that make sense? So think about you have a binary odometer on your car, and you want to take a mile off, okay, because you're at, let's say, one followed by 15 zeros. If you back it up, you expect all of these to be demoted not to nine, but to one. So, as far as a representation is concerned, it's one less the two to the fifteenth. Makes sense? And that number is two to the fifteenth minus one, which I'm not going to figure out what it is. Okay? But you get the jest of what's' going on here?

Okay. So that's enough. There is a little bit to be said about this bit right here. If I wanted to represent the numbers zero through two to the sixteen minus one, I could do that. Okay, that's two to the sixteenth different values. I don't want to say that negative numbers are as common as positive numbers, but they're not so uncommon that we don't want to have a contribution of the mapping to include negative numbers. So what usually happens is that this bit, right there, had nothing to do with magnitude. Okay, it's all about sign, whether or not you want a zero there because it's positive, or a one for negative. And that's usually what zero and one mean when they're inside a sign bit. Makes sense? Okay. So if I write down this and I have, let's say, four zeros followed by zero, one, one, one, okay? That's a seven. If I put all zeros there, it happens to be a seven that hogged a

little bit more memory, okay? It was a seven character initially, and now it's a seven short. I could argue that this would be the best way to represent negative seven. And you can look at it and you can recover the magnitude based on what's been drawn. And then just say – look all the way to the left – and say that one is basically the equivalent of a minus sign. That would be a fine way to do it if you wanted to go to the effort of actually always looking at the left-most bit to figure out whether it's negative or not. The reason it's not represented this way is because we want addition and subtraction to actually follow very simple rules, okay? Now, let me just be quite obtuse about how you do binary addition. Not because it's difficult, but because it's interesting in framing the argument as to why we adopt a different representation for negative numbers. Let's just deal with a four-bit quantity, okay? And I add a one to it. Okay. Binary addition is like decimal addition with more carries because you just don't have as many digits to absorb magnitude, so one plus one is two, but you write that down as a zero and you carry the one. You do this. And that's how seven plus one becomes eight. Okay. I imagine everybody followed that.

However, I want the computer hardware and its support for addition and subtraction to be as simple and obvious as possible. So what I'd like to do is have the number for positive seven, when added to the representation for negative seven, to very gracefully become all zeros. Does that make sense? Well, if I use the same rules, one – I'm sorry, zeros followed by zero, zero, one, one, one. This is four of them. This is eight of them. And I want to add that to seven zeros followed by four zeros, zero, one, one, one. Let's put a four in there. Let's put an eight in there. If I followed the same rules – and think about – I mean it's not like – the hardware is what's propagating electrons around and voltages around to emulate addition for us. If we want to follow the same rules, we would say, "Okay. Well, that's naught two. Carry the two. That's three. Carry the one. That's that." Let me just make sure I don't mess this up. Seven plus seven is fourteen, so it would be that right there. Okay. And then you'd have 11 zeros followed by a one. If I really just followed the algorithm I did up there blindly, that's how I would arrive at zero. Okay. And that's obviously not right. If I were to argue what representation that is, if this is negative seven, then this has to be negative fourteen. That's not what 7 plus negative 7 is. Okay. So that means that this right here, as negative number, has to be engineered in such a way that when you add this to this using normal binary ripple add pattern, okay, that you somehow get 16 zeros right here, okay? It's easier to not worry about how to get all zeros here. It's actually easier to figure out how to get all ones here. So if I put down four, five, six, seven, eight. One, two, three, four, let's mix it up. Let's put the number 15 down. And I want to figure out what number – or what bit pattern, to put right here to get all ones right here, then you know you would put a bit pattern right there that has a one where there's a zero right here and a zero where there's a one up here, okay? This right here is basically one mile away from turning over the odometer, okay? Does that make sense? Okay. So what I want to do is I want to recognize this as 15 and wonder whether or not this is really close to negative 15.

And the answer is, yes, it is because if I invert – take this right here and I invert all of the bits, if I add one to that number right there, do you understand I get this domino effect of all of these ones becoming zeros? I do get a one at the end, but it doesn't count because

there's no room for it. It overflows the two bytes, okay? So this right here would give me this one that I don't have space to draw because I'm out of memory, all the way down. So what I can do is rather than just changing this right here to be a sign bit, I can take the forward number, the positive 15, invert all the numbers, and add one to it, okay? Does that make sense? And that's how I get a representation for negative 15. This right here, this approach – it's what's called ones' complement – it's not used because it screws up addition. This notation for inventing the representation of the negative is what's called two's complement, okay? It's not like you have to memorize that. I'm just saying it. And this is how you've got all zeros. This is positive 15. This is negative 15. That is zero right there, okay? Does that make sense? The neat thing about two's complement is that if you have a negative number, and you want to figure out what negative of negative 15 is, you can follow the same rules. Invert all the bits – there, one – and add one to it, okay? And that's how you get positive 15 back. So this is nice symmetry going on with the system, okay? Make sense? Okay. Now, why am I focusing on this? Because you have to recognize that certainly in the world of characters and shorts, which is all we've discussed so far, that every single pattern corresponds to some real value in our world, okay? Characters, it's one of 256 values. We can fill in the Ascii table with ampersands and As and periods and colons and things like that, and have some unique integer be interpreted as a character every single time. As long as it's constant and it always maps to the same exact pattern, then it's a value mapping. As far as shorts are concerned, I could have used all 16 bits to represent magnitude. I'm not going to do that because I want there to be just as many negative numbers represented as positive numbers.

So I do, in fact dedicate all of the bits from that line to the right to magnitude, okay, and I use the left one – the left-most bit to basically communicate whether the number is negative or not, okay? That means that since there are two to the sixteenth different patterns available to a two-byte figure, that the short can distinguish between that many values. Rather than having it represent zero through two to the sixteenth minus one, I actually have it represent negative two to the fifteen – I'm sorry, negative two to the fourteen – I'm sorry, negative two to the fifteenth through two to the fifteenth minus one. Does that make sense? And everything's center around zero. So I have just as many representations for negative numbers as I have for positive numbers. Okay? Makes sense? Okay. So let's start doing some really simple code, not because it's code you'd normally write. Sometimes you do, not very often. But just to understand what happens when you do something like this. I have a CAR variable, CH, and I set it equal to capital A. And then I have an int variable called – actually, let me make it a short – S, and I set it equal to CH. You don't need a cast for that. What you're really doing is you're just setting S equal to whatever number is backing CH. There's a question right there?

Student:[Inaudible]

Instructor (Jerry Cain):All right. It shouldn't have been. Oh, I just – I'm sorry, I inverted the bit pattern, and then I said you would add one to this, and I just didn't change the bit in the drawing. Where'd you go? I just saw – okay. So I didn't add one to this yet. But in the conversation at the time I thought it was clear. Okay. Does this make sense to people? Okay. There's – certainly it's gonna compile and it's going to execute. And based

on the other seven boards I've drawn in, you should have some idea as to what's gonna happen in response to this line. Print out is the equivalent of a cout statement, but it's in pure C. And if I want to print out a short – actually, let me just cout. Less than, less than S is less than, less than PNDL. In response to that, I expect it to print out the number 65. So to the console I would expect that to be printed. Why is that the case? Because the declaration of CH doesn't put a capital A there, it puts the integer value that backs it there, which I will draw as decimal. You know that it's really ones and zeros. And so when time comes for you to assign to S, what happens in order to achieve the effect of the assignment, it will copy this bit pattern. And this is what it really does electronically. It just replicates whatever bit pattern is right here onto that space right there. It smears these bits onto this little byte like it's peanut butter. And puts a 65 down there. And all the extra space is just padded, in this case, with zeros. So that's how 65 goes from a one-byte representation to a two-byte representation. Does that make sense? I simplified this a little bit. When I put a 65 down here and a smear of 65 in there, I happen to know that the left-most bit is a zero there. It's a positive number so that shouldn't surprise you, okay? Does that make sense to people? What happens if I do the opposite direction? I do this int – I'm sorry, we're not at ints yet. Short – completely new program – S is equal to, I'll say 67, and I do this. It compiles. There's no casts needed. As far as how the 67 is laid down, it's zero, one, zero, zero, zero, zero, one, one. It's two more than 65, obviously. It has an extra byte of all zeros. And this is S.

So when CH gets laid down in memory, and it's assigned to S, two bytes of information, and 16 bits cannot somehow be wedged economically into an eight-bit pattern. So what C and C++, and many program languages for that matter, do is they simply punt on the stuff they don't have room for. And they assume that if you're assigning a large number to a smaller one, and the smaller one can only accommodate a certain range of values in the first place, that your interested in the minutia of the smaller bits. Does that make sense to people? So what happens is it replicates this right here, and it punts on this. And this is how, when you do this right here, okay, you go ahead and you print out a C. Make sense to everybody? Okay. Now, I've kind of evaded the whole negative number things, but negative values don't work too well with characters because unsigned CARs – most characters are unsigned. So you actually do get all positive values with the representations. You know enough about shorts to know that the two-byte figures – I've already told you that longs and ints, at least in our world, are four bytes. They're just a four-byte equivalent of a short. So let me deal with this example. I go ahead and I do a short, S is equal to, I'll just say – let me write it this way. No, I'll just write it as two to the tenth plus two to the third plus two to the zero. That is, of course, not real C. But I'm just writing it because I want to be clear about what the bit pattern for that number is. So just think about whatever number that adds up to as being stored in S. Okay. This is two to the eighth, two to the ninth. One, zero, zero, preceded by all zeros. Lots of zeros. One, zero, zero, one. If I take an int, i, and I set it equal to S, the same argument that I made in the CAR to short assignment can be taken here. And this is how – and this is somehow less surprising because both of them represent integers.

This is all zeros. All zeros. Lots of zeros followed by one, zero, zero, zero, zero, zero, zero, one, zero, zero. And that's why. You just have a lot more space to represent the

same small number, okay? Trick question. If I set int i equal to – I have 32 bits available to me to represent pretty big numbers, so I'm gonna do this. Two to the twenty-third plus two to the twenty-first plus two to the fifteenth plus, let's say, seven. Okay? And I'm being quite deliberate in my power of two representation of these numbers because seven always means that at the bottom, okay? Two to the fifteenth means there's a one right there. Two to the – actually, let me change this to two to the fourteen. Make this a zero, one. Two to the twenty-first – although this is two to the twenty-fourth. Two to the twenty-third, followed by zeros. All zeros right there. So that's more or less what the bit pattern for that, as a four-byte integer would look like. I go ahead and I set short S equal to i. You could argue that wow, that numbers so big it's not gonna fit in the short, okay? And so you might argue that well maybe we should try and come as close as possible and make S the biggest number it can be so it can try really hard to look like this number. And that's not going to happen. It's gonna do the simplest thing. Remember this is implemented electronically, and every single example over there has more or less been realized by just doing a bit pattern copy, okay? If you're writing this way, you probably know that you're going and taking a four-byte quantity and using it to initialize a two-byte quantity. So lay that down, this is S. And all it does is say, "You know what, I have no patience for you right there. You're out. I'm just gonna copy this down." Okay? And so I do this followed by lots of zeros, followed by lots more zeros, followed by one, one, one. And I print out S. I'm gonna get the number that is two to the fourteenth plus seven. Does that make sense to people?

Okay. So let me go back and do one more example before I move on to floating-point. Oh, yeah?

Student: Initially you had three to the fifteenth?

Instructor (Jerry Cain):Right. I'd say it's – it's actually confusing as to what happens. It certainly is. I actually don't know what happens when what is a magnitude bit actually becomes a sign bit. I have to say I certainly should know what happens. I just don't, which why I gracefully said, "Oh, I have an idea. Let me just change this to two to the fourteenth." I'll actually run this remnant after lecture and I'll just mail the class this as part of this email that everyone is getting today, okay? Yep?

Student:[Inaudible] the other way around [inaudible] a sign short [inaudible]?

Instructor (Jerry Cain):Well, it will always preserve sign, and I'm gonna – that's the very example I'm gonna do right now, okay? Suppose I did this. Short S is equal to negative one. Totally reasonable. Do any of you have any idea what the bit pattern for that would look like? And you can only answer if you didn't know the answer prior to 11:00 a.m. today. Okay. I want to be able to add one to negative one and get all zeros, okay? Does that make sense? So the representation for this is actually all ones. In fact, anytime you see all ones in a multi-byte figure, it means it's trying to represent negative one. Why? Because when I add positive one to that, it causes this domino effect and makes all those ones zeros. Does that make sense? So to answer your question, int i equal to S, logically I'm supposed to get int to be this very spacious representation of negative

one. It actually does use the bit pattern copy approach. It copies these. I've just copied all the magnitude, okay? And by what I put down there, it's either one or – I'm sorry. It's either a very large number or it's negative one. We're told that it's negative one right there, okay? What happens is that when you assign this to that right there, it doesn't just place zeros right there because then all of the sudden it would be destroying the sign bit. It would be putting the zero in the sign bit right there. Make sense? So what it really does, and it actually did this over here, but it was just more obvious, is it takes whatever this is in the original figure and replicates that all the way through. If these would have otherwise been all zeros, and I want to be able to let this one continue a domino effect when you add a positive number to a negative number, you technically do what's called "sign extend" the figure with all of these extra ones. So now you have something that has twice as many dominos that fall over when you add positive one to it, okay? Does that make sense?

Okay. So there you have that. As far as character shorts, ints, and longs, they're all really very similar in that they some binary representation in the back representing them. They happen to map to real numbers, for ints, longs, and shorts. They happen to pixelate on the screen as letters of the alphabet, even though they're really numbers, very small numbers in memory, okay? But the overarching point, and I don't want you to – I actually don't want you to remember – memorize too much of this. Like, if you know what – if you know that seven is one, one, one, and you know that all ones is negative one, that's fine. I just want you to understand the concept with integers I have four bytes. I have 32 bits. That means I have two to the thirty-second different patterns available to me to map to whatever subset of a full integer range I want. The easiest thing to do is to just go from two to the negative thirty-first through two to the positive thirty-first minus one, okay? There's zero in the middle. That's why it breaks it symmetrically a little bit. When I go and start concerning myself with floats – I – you're probably more used to doubles, but this is just a smaller version of doubles. I have four bytes available to me to represent floating-point numbers, integers with decimal parts following it, in any way I want to. This isn't the way it really works, but let me just invent an idea here. Pretend that this is how it works. We're not drawing any boxes yet. I could do, let's say I have a sign bit. I'll represent that up here as a plus or minus. And if I have 32 bits, you'd, by default, thinking about bits and contribution of two to the thirtieth, two to the twenty-nine, all the way down through some contribution of two to the zero. And I'm just describing all the things that can adopt zeros or ones to represent some number, okay? But I want floats to be able to have fractional parts.

So I'll be moving in the fractional direction, and say, "You know what? Why don't I sacrifice two to the thirtieth, and let one bit actually be a contribution of two to the negative first?" I'm just making this up. Well, I'm not making it up. This is the way I've done it the last seven times I taught this. But I'm moving toward what will really be the representation for floating-point numbers. If I happen to have 32 bits right here. And I lay down this right here. That's not the number seven – I'm sorry, that's not the number 15 anymore. Now, it's number seven point five. Does that make sense? Okay, well floats aren't very useful if now all you have are integers and half-integers. So what I'm gonna do is I'm gonna stop drawing these things above it because I have to keep erasing them. Let's

just assume that rather than the last bit being a contribution of two to the negative first, let me let that be a contribution of negative two to the negative first, and that let that be a contribution of two to the negative two. Now I can go down to quarter fractions. Does that make sense? Well, what I could do is I could make this right here a contribution of two to the zero, two to the negative one, two to the negative two, three four, five, six, seven, eight, two to the negative nine. And if I wanted to represent Pi – I'm not going to draw it on the board because I'm not really sure what it is, although I know that this part would be one, one – then I would use the remaining nine bits that are available to me, okay, to do as good a job using contributions of two to the negative first, and two to the negative third, and two the negative seventh to come as close as possible to point one four one five whatever it is, okay? Does that make sense to – I'm assuming? It is an interesting point to remember that because you're using a finite amount of memory, you're not going to do a perfect job representing all numbers in the infinite, and infinitely dense, real number domain, okay? But you just assume that there's enough bits dedicated to fractional parts that you can come close enough without it not really impacting what you're trying to do, okay? You only print it out to four decimal places, or something that just looks like it's perfect, okay? Does that make sense? It turns out if I do it that way, then addition works fine. So I add two point five contributions and it ripples to give me a one and I carry a one. It just words exactly the same way. Does that make sense? Okay. It turns out that this is not the way it's represented, but it is a technically a reasonable way to do it. And when they came up with the standard for representing floating-point numbers, they could have gone this way. They just elected not to.

So what I'm gonna do now is I'm gonna show you what it really does look like. It's a very weird thing. But remember that they can interpret a 32-bit pattern any way they want to, as long as the protocol is clear, and it's done exactly the same way every single time. So for the twentieth time today, I'm gonna draw a four byte figure. I'm gonna leave it open as four byte rectangle because I'm not gonna subdivide it into bytes perfectly. I'm going to make this a sign bit because I do want to represent – I want negative numbers and positive numbers that are floating-point to have an equal shot at being represented, okay? That's one of the 32 bits. Does that make sense? The next eight bits are actually taken to be a magnitude only – I say it that way. I should just call it an unsigned integer – from here to there, okay? And the remaining 23 bits talk about contributions of two to the negative one, and two to the negative two, and two to the negative three. Okay, this right here, I'm gonna abbreviate as EXP. And this right here, I'm just gonna abbreviate as dot XXX XX, okay? The – what – this figure and how it's subdivided is trying to represent this as a number. Negative one to – I'll abbreviate this as S – to S right there. One point XXX XX times two to the one twenty-eight – I'm sorry, hold on a second. EXP minus one twenty-seven, okay? It's a little weird to kind of figure out how the top box matches to the bottom one. What this means is that these 23 bits somehow take a shot at representing point zero, perfectly as it turns out, to something that's as close to point nine, nine bar as you could possibly get with 23 bits of information. When these are all ones, it's not negative one. It's basically one minus two to the twenty-third. Does that make sense to every? Okay. That is added to one to become the factor that multiplies some perfect power of two. Okay? This right here ranges between two to the eighth – I'm sorry, 255 and zero. Does that make sense?

When it's 255 and it's all ones, it means the exponent is very, very large. Does that make sense? When it's all zeros, it means the exponent is really small. So the exponent, the way I've drawn this here, can range from 128 all the way down to negative 127. Makes sense? That means this right here can actually scale the number that's being represented to be huge, in the two to the one twenty-eight domain, or very small, two to the negative one twenty-seventh, okay? The number of added to the world down to the size of an atom, okay? You may think this is a weird thing to multiply it by, but because this power of two-thing right there really means the number is being represented in the power of two domain. You may question whether or not any number I can think of can be represented by this thing right here. And then once you come up with a representation, you just dissect it and figure out how to lay down a bit pattern in 32 byte – 32-bit figure. Let me just put the number seven point zero right there. Well, how do I know that that can be represented right here? Seven point zero is not seven point zero. It's seven point zero times two to the zeroth, okay? There's not way to get and layer that seven point zero over this one point XXX and figure out how – what XXX should be. XXX is bound between zero and point nine bar. But I can really write it this way. Three point five times two to the first, rather one point seven five times two to the second. So as long as I can do a plus or minus on the exponent, I can divide and multiply this by two to squash this into the one to one point nine range. And just make sure that – I have to give up if this becomes larger than 128 or less than negative 127. But you're dealing with, then, absurdly large numbers, or absurdly small numbers. But doubles love the absurdity because they have space for that accurate of a fraction, okay? Does that make sense to people? Okay, so this right here happens to be the way that floating-point numbers are actually represented in memory. If you had the means, and you will in a few weeks, to go down and look at the bit patterns for a float, you would be able to pull the bit patterns out, actually write them down, do the conversion right here, and figure out what it would print at. It would be a little tedious, but you certainly could do it. And you'd understand what the protocol for coming from representation to floating-point number would be, okay? Let me hit the last ten minutes and talk about what happens when you assign an integer to a float, or a float to an integer, okay? I'm gonna get a little crazy on you on the code, all right. But you'll be able to take it.

I have this int i is equal to 35 – actually, let me chose a smaller number. Let me do just five is fine. And then I do this. Float F is equal to i. Now you know that this as a 32-bit pattern had lots of zeros, followed by zero, one, zero, one at the end, four plus one, okay? Makes sense? When I do this, if I print out F, don't let all this talk about bits and representation confuse the matter. When you print out F there, it's going to print the number five, okay? The interesting thing here is that the representation of five as a decimal number is very, very different than the representation of five using this protocol right here. So every time – not that you shouldn't do it – but every time you assign an int to a float, or a float to an int, it actually has to evaluate what number the original bit pattern corresponds to. And then it has to invent a new bit pattern that can lay down in a float variable. Does that make sense? This five is not – the five isn't five so much as it is one point two five times two to the second. Okay, as far as this is concerned right here. So that five, when it's really interpreted to be a five point zero, it's really taken to be a one point two five – is that right? Yeah. – Times two to the second. So we have to choose

EXP to be 129 and we have to choose XXX to be point two five. That means when you lay down a bit pattern for five point zero, you expect a one to be right there. And you expect one – one, zero, zero, zero, zero, zero, zero, one to be laid down right there, 128 plus one. Does that make sense to people? You gotta nod your head, or else I don't know. Okay. This is very different – and this is where things start to get wacky – and this is what one oh seven's all about. If I do this right here, int i is equal to 37. And then I do this, float F is equal to asterisk – you're all ready scared. Float, star, ampersand of i. I'm gonna be very technical in the way I describe this, but I want you to get it. The example above the double line, it evaluates i, discovers that it represents five, so it knows how to initialize F. Does that make sense? This right here isn't an operation. It doesn't evaluate i at all. All it does is it evaluates the location of i. Does that make sense? So when the 37, with it's ones and zeros represented right there, this is where i is in memory. The ampersand of i represents that arrow right there, okay? Since i is of type int, ampersand of i is of type int, star, raw exposed address of a variable. That's four bytes that happens to be storing something we understand to be an int. And then we seduce it, momentarily, into thinking that it's a float star, okay?

Now, this doesn't cause bits to move around, saying, "Oh, I have to pretend I'm something else." That would be i reacting to an operation against the address of i, okay? All the furniture in the house stays exactly the same, okay? All the ones and zeros assume their original position. They don't assume, they stay in their original position. It doesn't tell i to move at all. But the type system of this line says, "Oh, you know what? Oh, look. I'm pointing to a float star. Isn't that interesting? Now, I'm gonna be reference it." And whatever bit pattern happened to be there corresponds to some float. We have no idea what it is, except I do know that it's not going to be thirty-seven point zero, okay. Does that make sense? In fact it's small enough that all the bits for the number 37 are gonna be down here, right, leaving all of these zeros to the left of it, okay? So if I say stop and look at this four byte figure through a new set of glasses, this is going to be all zeros, which means that the overall number is gonna be weighed by two to the negative one twenty-seven. Makes sense? There's gonna be some contribution of one point XXX, but this is nothing compared to the weight of a two to the negative one twenty-seven. So as a result of this right here, and this assignment, if I print out F after this, it's just gonna be some ridiculously small number because the bits for 37 happen to occupy positions in the floating-point format that contribute to the negative twenty-third, and to the negative twentieth, and things like that, okay? Does that make sense to people? Okay. So this is representative of the type of things that we're gonna be doing for the next week and a half. A lot of the examples up front are going to seem contrived and meaningless. I don't want to say that they're meaningless. They're certainly contrived because I just want you to get an understanding of how memory is manipulated at the processor level.

Ultimately, come next Wednesday, we're gonna be able to write real code that leverages off of this understanding of how bits are laid down, and how ints versus floats versus doubles are all represented, okay? I have two minutes. I want to try one more example. I just want to introduce you to yet one more complexity of the C and C++ type system, and all this cast business. Let me do this. Let me do float F is equal to seven point zero. And let me do this short S is equal to asterisk, short star, ampersand of F. Looks very similar

to this, except there's the one interesting part that's being introduced to this problem, is that the figures are different sizes, okay? Here I laid down F. It stores the number seven point zero in there. And that's the bit pattern for it, okay? The second line says, "I don't care what F is. I trust that it's normally interpreted as a float, and that's why I know that this arrow is of type float, star." Oh, let's pretend – no, it isn't any more. You're actually pointing – that arrow we just evaluated? It wasn't pointing to a float. We were wrong. It's actually pointing to a two byte short. So all of the sudden, it only sees this far, okay? It's got twenty-forty vision, and this right here, this arrow, gets dereferenced. And as far as the initialization of S is concerned, it assumes that this is a short. It assumes that this is a short so it can achieve the effect of the assignment by just replicating this bit pattern right there, okay? And so it gets that. Okay, and whatever bit pattern that happens to correspond to in the short integer domain, is what it is. So when we print it out, it's going to print something. Seven point zero means that there's probably gonna be some non-zero bits right here. So it's actually going to be a fairly – it's gonna have ones in the upper half of the representation. So S is gonna be non-zero. I'm pretty sure of that, okay? Does that make sense to people?

Okay. That's a good place to leave. Come Monday, we'll start talking about – we'll talk a little bit about doubles, not much. Talk about strucks, pointers, all of that stuff, and eventually start to write real code. Okay.

[End of Audio]

Duration: 51 minutes