ProgrammingParadigms-Lecture11

Instructor (Jerry Cain):Hey, everyone. Welcome. I actually have one handout for you today, and Ryan just walked in with it. So he's going to be passing it around to the people who are in the room. Remember, we have a special discussion section today, at – I'm forgetting what time it is. What did I say? 1:15 p.m.? I'd check.—whatever I said on Wednesday. What did I say? 2:15pm, okay. I don't know why I'm forgetting: 2:15pm to 3:05pm, in scaling 191. We're having a special discussion section, just this one week because I want some example problems to be in – to sit in front of you, and for you to think about them, more than two days before the next assignment deadline. Which is why I'm having it today.

It's going to be videotaped. I'm gonna call SCPD after the lecture today, and try and make sure that they post it by 5:00 p.m. today, so it's around for the weekend, online, so you can watch it. But I've already, I think, pretty well advertised it, Assignment 4; I think to be kind of the biggest surprise in terms of workload and difficulty for most students. So that's why I'm kind of advertising it now, to be something you want to start on, sooner than later. Because if you start next Thursday, it's very likely, you will not finish on time. So just try and get – take a stab at it a little bit ahead of time, this time.

When I left you last time, I had shown you two examples of how function call and return works, in general, but specifically in our assembly language. So what I want to do is, I want to do one specific example, actually much, even simpler than the last example I did. Because I want to make a transition from C code generation to C + + code generation, and show you just how similar the two languages ultimately end up looking like, when they compile to assembly code.

So let's deal with this scenario: a void function. I just want to say foo, and I'll concern myself with the allocation of two local variables. I'll set x equal to 11; I'll set y equal to 17. And then, I'm going to call this swap function. This time, I'm interested in writing swap to show you what the assembly code looks like. And I think you'll be surprised to see how the pointer version of swap and the reference version of swap look exactly the same. But the way I'm writing it, right now, this is pure C code. Actually, you can call it C + + code, for that matter. And then, that's all I want to do.

I just want you to understand how code for this would be generated. I don't have any parameters, whatsoever, so I would expect the stack pointer, right there, to have the safe p c associated with whatever function happens to be calling foo, right here. There's a call foo instruction in someone else's assembly stream, and that safe p c is the address of the instruction that comes directly after that. That's the trail of popcorn I was talking about, on Wednesday. The first thing that'll happen is that this thing will actually make space for its two locals. I actually like getting in the habit of kind of taking care of my deallocation call right away. That make sense to people? Okay? Well, as soon as I write that, I just want to remember just so I just don't forget to put it on the page. I like to put the balancing s p change at the bottom of the stream. Now, I'll concern myself with this, right here. This brings this down, to introduce those 8 bytes. X is above y, so this

instruction right there, translates to m of s p plus 4 is equal to 11; then m of s p is equal to 17, and then virtually all of the rest is associated with the work that's necessary to set up and react to – set up a function call, and react to its return. Okay? So what I want to do here is I want to evaluate ampersand of x, and ampersand of y. The fact that they're an eleven here – oops – an eleven there and a 17, is really immaterial to the evaluation of these two things, right here. This, given that setup right there, is synonymous with a p c plus 4; this is synonymous with p c. Okay? I want to put those values in registers: R 1 is equal to s p – I'm sorry, that's right – s p y 2 is equal to s p plus 4. This is ampersand of y; this is ampersand of x. I want to further decrement the stack pointer by minus 8. The fact that this is minus 8 is just a coincidence with that being minus 8. There happen to be two 4-byte parameters being passed here; there were two 4-byte parameters – 2 4-byte locals, local to foo. Once I've done that, I've decremented this by 4 more bytes. That marks the bottom of the foo activation record. I'm now building a partial activation record for the swap function. I want to lay this as a figure, right there. This, as a figure, right there. This first parameter, the zero parameter, actually goes at the bottom. So I want to do this: m of s p, which just changed, is equal to r 2; m of s p plus 4 is equal to r 1. Okay? That, basically, gets this right here, to point to that, and this to point to that. So when we go ahead, and call this swap function, we're just inferring its prototype to take two int stars, it just sees this. Technically, it has the addresses, the locations of the two ints it's going to be dealing with, but it doesn't, technically, know that they're right above it on the stack frame. It actually just has the addresses on their little houses that just happen to be just down the block. Okay?

This, right here, all it has to do is basically clean up the parameters. When the call swap function happens, p c is pointing right there. It can, by protocol, assume that that's where the safe p c is left – I'm sorry, the stack pointer is left, when it jumps back to this instruction because the return at the end of the swap is responsible for bringing it all the way back up here, just by protocol. Does that make sense? Okay.

So all we need to do: fill in the space. This balances that; that balances that. You could coalesce these to one statement, if you wanted to. I just don't see a compelling reason to. And then, since there's nothing being done with the new values of x and y, I can just return. Okay? Does that make sense to people? As far as the code for swap is concerned, this is void swap. I'll write it to be int star specific; a p int star b p int temp is equal to asterisk a p. There's actually a little bit of work here, at the assembly code level. I know you've seen this implementation, probably 40 times, in the last two quarters, but there's some value in actually seeing it, one final time, and for me to generate assembly code for it.

It starts executing with the assumption that s p is pointing to a safe p c. In this particular example, it happens to be that address, right there, that's the safe p c. This is internally referred to as a p, b p. It's not like the words a p and b p are written on the hardware, but there's just – the code is generated, in such a way, that a p is always synonymous, at the moment, with s p plus 4; and b p is synonymous with s p plus 8. Okay? The moment the stack pointer points right there, I have to make space for my one local variable. That happens because I do this. I'll make it clear that this is associated with the label swap that

we actually call or jump to. That brings this down here. This is locally referred to as temp in the code; this is the space that corresponds to temp in the activation record. The offsets of a p and b p actually shift, a little bit. Now, this is an address, s p plus 8, s p plus 12. I have to rely on these things pointing to real data. I don't need to know where that data is to write the correct assembly code. I need to evaluate a p and then, what a p points to. So as part of initializing that, right there, I have to load into a register, m of s p plus 8. Do you understand that it lifts this bit pattern, right there, and places it in a register, so I can basically do a cascade of dereferences. Does that make sense to people? Okay? Now I have a p in the register, I can do this and now I have the very integer that is addressed by this a p thing, okay, sitting inside r 2. You may ask whether or not you can do something like this. Conceptually, of course you can do that, except you're not going to find an architecture that supports double cascade in a single assembly code instruction. Okay? So that's why they're broken up over several lines.

So let's assume that this points to that, right there, that the integer that has to be loaded into temp. That's only going to happen because I do this. Oops. And that ends this line, right here. Does that make sense to people? Yeah, go ahead.

Student: When you say r2 equals m, r of r 1 –

Instructor (Jerry Cain):Yep.

Student: Why does it not just copy the address in, say, a p and actually [inaudible]

Instructor (Jerry Cain):Well, the memory of – you basically go inside an array of bytes that's based addressed, right there. Okay? So you think about the entire array of – entire array of bytes that's in RAM as being indexed from position zero. And so when you put that number inside, it actually goes to that address. We know that s p plus 8, this right here, we know that the 4 byte bit pattern that resides there is actually the raw address of some integer. The assembly code doesn't know that, but we do. So we're asking it to shovel around a 4-byte figure and put it into this temp variable. Right? This loads that address. This address right here, let's say it's address 4,000. I just loaded a 4,000 into r 1. Maybe it's the case that the 17 is in address 4,000. Okay, that would make sense. So by effectively doing m of r 1, in this particular case, I'm doing m of 4,000 because that's what r 1 evaluates to. Okay? And by default, we always do it with 4-byte transfers. Okay? So this would get a 17 to play the role of happy face, and would put a 17 right there. Does that make sense? Okay. So now, what has to happen is something similar. I have to do this right here. But it's actually a little different than this line right here because I have to find the L value by actually following a pointer. The L value, being temp, is directly on the stack frame, in this case. Right? The space where the new value is being written is actually one hop off of the stack frame. Okay? Let me evaluate asterisk b p first because it's very similar. I will do r 1 is equal to m of s p plus 12. That loads just the value of b p because that's what's stored as a variable on the stack frame. Okay? In r 2, I'll do m of r 1, but r 1 has a different value this time. Maybe it is the flat emoticon, okay, and that gets loaded into the r 2 register. That's what actually has to be written over this smiley face, right there. Make sense? So in order for that to happen, I have to load

this again: r 3 is equal to m of s p plus 8. And I don't want to load m of r 3 because I don't care about the old value. I want to actually want to set m of r 3 equal to r 2. Okay? Does that sit well with everybody? Just making sure I did that right. Yeah, a p right there. Okay? Making sense? The last line is actually very similar to the first one. I know that temp is sitting right at the base of the entire activation record. Now, what I have to do is I have to load the address associated with b p into r 2. That's stored at m of s p plus 12. And then I have to do m of r 2 – I'm sorry, yeah – m of r 2 is equal to r 1. That realizes this, right here. So in those three blocks, I've achieved the algorithm that is the rotation of these three—of these two-byte patterns, actually, through a third. Right before I clean this up here, I have to do an s p is equal to s p plus 4. Internally, I have to bring the stack pointer up to the place where it was before I executed any code for this function. Okay? That means that s p is now pointing to the safe p c. Which is perfect because in response to this return instruction, our return actually knows, just procedurally, to go ahead and pull that value out, populate it with the – place it into the real p c register and simultaneously, or just after that, bring this back up to there. Okay? That's exactly where the s p pointer was pointing before that call statement. Does that make sense? Okay.

So I went through that exhaustively because pointers are still mysterious for some people, and understandably, because they're difficult. So if you start to understand them, at the hardware level, you have some idea as to what the pointers are trying to achieve. They really are raw locations of figures being moved around, or at least, manipulated. Okay? What I want to do now is, I want to show you what the activation record and the function call and return mechanisms would be for the C + + version of swap. This is very close. I'll write it again and leave it as an open question, for about two minutes, as to what the activation record must look like. I'll just do a int ampersand b. So just pretend we're dealing with pure C, except that we've integrated the reference feature from C + +. Okay? Int temp is equal to – what did I see over there?—a, and a is equal to b, and then b is equal to temp. Algorithmically, it looks similar, and it even sort of has the same side effect. But you may question how things are actually working behind the scenes, in order to swap things by reference, as by opposed address. Let me draw the activation record for you. This is the thing that's referred to as a; this is the thing that's referred to as b. This is always below that. That's just the rule for parameters to a function or a method call. There's a safe p c, right here, pointing to the instruction that comes right after the call to the swap function. And then, ultimately, the very first instruction does an s p is equal to s p minus 4. So that, this is the entire activation record that is in place before swap actually does anything in terms of byte moving around. Okay? This a and this b, just because we refer to them as if they're direct integers doesn't mean that, behind the scenes, they have to actually be real integers. The way pointer – I'm sorry – the way references work is that they are basically just automatically dereferenced pointers. Okay? So without writing assembly code for this, when I do this, and I do this, just because I'm passing in x and y – that's the way you're familiar with the swap function from 106b and 106x – just because you're not putting the ampersands in front of those xs and ys, doesn't mean that compiler is blocked from passing their addresses.

This, right here, has to be an L value; this has to be an L value, as well. It means it actually has to name a space in memory, okay, that can be involved in an update or an

exchange by reference. When this, right here, is compiled to cs107 assembly language, it actually does exactly the same thing that that does, right there. C + + would say, “Oh, this is x and y, but I’m not supposed to evaluate x and y because I can tell, from the prototype, that they’re being evaluated – they’re being passed by reference.” So the way it does it is, on your behalf, just secretly says, “Okay, they need – they really need this x and this y to be updated in response to the swap call.” That’s only gonna happen if this, as a function, knows where x and y are. Okay? So the illusion is that a and b, as stand-alone integers, are second names for what was originally referred to as x and y. Behind the scenes, what happens is that this lays down a pointer and this lays down a pointer. You don’t have to use the word pointer to understand references; you can just say it’s magic and somehow, it actually does update the original values. It lays down the address of these things. The assembly code that is generated for this function, right here, understands even though you don’t necessarily know this. It understands that it’s passing around pointers wherever references are expected. And so it automatically does the dereferencing of the raw pointers that implement those references, for you. The assembly code for this function is exactly the same as this, like, down to the instruction, down to the offsets; everything is exactly the same. Okay? Do you understand how that’s possible? Just because you don’t put ampersands in there, doesn’t leave the compiler clueless because it knows, from the prototype, that it’s expecting references. It’s just this second flavor of actually passing things around by address. You’re just not required to deal with the ampersands and the asterisks. Okay?

It knows because of the way you declared these, that every time you refer to a and b in the code, that you’re really referring to whatever is accessible from the address that’s stored inside the space for a. Okay? Does that sit well with everybody? Okay, that’s good. So people freaked out a little bit on Assignment 1, I think – or Assignment 2 – when they saw local variables declared as references. Like, you’re used to the whole parameter being a reference, as if this data type and that data type, you’re only allowed to put them in parameter lists. That’s just not true. If you want to do this, if you do this right here, then that’s just traditional allocation of a second variable that happens to be initialized to the x variable – I’m sorry – to whatever x evaluates to. Okay? And that’s just normal. If you want to do this, you can. Okay? And this isn’t a very compelling reason because ints are small, and there’s no algorithm attached to this code. So you’re not necessarily clear why it would do that. The was this is set up, is that x really is set aside as an integer, with a 17; y is really set aside with it’s own copy of the 17. But z is declared – you drew them in 106b and 106x, this way. And the picture is totally relevant to the actual compilation measures because it really does associate the address of y inside the space that’s set aside for z. Okay? Does that make sense? If I were to do this, you would totally draw that without the shaded box. Right? This and this, from an assembly code standpoint, are precisely the same. At the C and C + + language level, you’re required to put the explicit asterisk in here; you’re not there. Okay?

You may say, “Well, why would I ever use actual pointers, if references just become pointers?” Well, references are convenient, in the sense that they give the illusion of second names for original figures that are declared elsewhere. The thing is, with references, you can’t rebind the references to new L values. Do you understand what I

mean when I say that? It's a technical way of saying you can't reassociate z with x, as opposed to y. Once it's bound as a reference to another space, that's in place forever. So you don't have the flexibility to change what the arrow – where the arrow points to. You do have that ability with raw pointers. So you could not very easily build a link list, if you just had references. Does that make sense to people? Okay. So that's why the pointer still has utility, even in C + +. Okay? You saw a method where – I think it was, like, get last player, or something like that. There was some method, in one of the classes for Assignments 1 and 2, that returned a reference. If you were, like, "I've never seen that before; what does that mean?" All it's doing is it's returning a pointer behind the scenes, okay, and you don't have to deal with it that way. You can just actually assume that it's returning a real string, as opposed to a string star, or an int as opposed to an int star, and just deal with it like that. But behind the scenes, it's really referring to an int or a string that resides elsewhere. Does that make sense to people? Yes, no? Okay. So even though in C – when you – C + +, you start dealing with references, it's not like the compilation efforts that are in place. And the assembly code that's generated is fundamentally different in the way it supports function call and return, and just code execution. Okay? It just has a different language semantic at the C + + level that happens to be realized, similarly, at the assembly code level. Okay?

References are the one addition to C + +, I'm sorry, there's a few –but there are – a lot of people, when they program in C + +, they actually program in C, where they just happen to use objects and references. They don't use inheritance; they don't necessarily use templates all that often, although most of the time, they do. But that's not an object-oriented issue. They don't use inheritance; they don't use a lot of the clever little C + +isms that happen to be in there. They really just code in C, with references and objects. And they just think of references as more convenient pointers, less confusing. And they think of objects as structs that happen to have methods defined inside. Does that mean there's a reasonable analogy I'm throwing by, I'm assuming? Okay? Well, when you program as an LL purist, you're not supposed to think of objects as structs, you're supposed to think of them as these alive entities, that actually are self-maintaining and you communicate with them through the series of instructions that you know they they'll respond to because you read the dot h file. Okay? Turns out that structs and classes – just looking at this from an "under the hood" perspective – structs and classes are laid down in memory virtually the same way. Not even virtually, exactly the same way. Okay? In C + +, structs can have methods. The structs – I'm not misusing a word there – structs can have constructors, destructors, and methods, as can classes. Classes aren't required to have any methods. The only difference between structs and classes, in C + +, is that the default access modifier for classes is private, and the default access modifier for structs is public. Does that make sense to people? Okay? So the compiler actually views them as more or less the same thing. It's just there's a little bit of basically a switch statement at the beginning, where it says, "Was it declared as a struct or a class?" And then, it says, "Okay, it's either private or public, by default." Okay?

When you do something like this: class, I'm gonna say – I'll just do binky, and I'll worry about the public section in a second. Let's not worry about constructors or destructors. They're interesting, but they're just complicated, so I want to blow them off for a minute.

And then, privately, let's say that I have an int called winky, a char star called blinky, and let's say I have a wedged character array of size 8, called slinky. And let's just get all these semicolons to look like semicolons. And that is it. And those are the only data fields I push inside. Every single time you declare one of these binky records, or binky objects, you do this. It shouldn't be that alarming that you really get a blob of memory that packs all the memory needed for these three fields into one rectangle. And it uses exactly the same layout rules because of all the fields, winky is declared before whatever I called it, blinky. And then, on top of that, is an array of 8 characters called slinky, that this entire thing is the b type. It's laid out that way because if you look at this, and think of it as a struct, the layout rules are exactly the same. Winky is stacked at offset zero; this is stacked on top of that, and this resides on top. This is an exposed pointer, which is why there 4 bytes for that rectangle. This is an array that's wedged inside the struct/class, which is why it looks like that for the upper 50 percent of it. When you do something like this – forget about constructors, let's just invent a method, right here. Let's just assume that I have this method, int and some other thing, donkey where I pass in – let's say a [inaudible] x and an int y. Okay? And let me actually declare another method, char star – I'm running out of consonants – minky, and all I want to do is I want to pass in, I'll say, an int star called z. And I will inline the implementation of this thing to do this: int y – I don't want to do that – int w is equal to asterisk z. And then, I want to return slinky plus whatever I get by calling this donkey method.

I'm not going to implement all the code for this, but I will do this: Winky, winky, and that's gonna be good enough. It's a very short method. You don't normally inline it in the dot h; I'm just putting all the code in one board. Okay? Just look at this from an implementation standpoint, and let me ask something I know you know the answer to. Z right there, is explicitly passed in as a parameter; w is some local variable, okay, that's clearly gonna be part of the activation record for this minky thing. Why does it have access to winky right there, or slinky right there? Because you know that it has the address of the object that's being operated on. Does that make sense? The this pointer is always passed around as the negative 1th argument, or the 0th argument, where everything is slided over to make space for the this pointer. It's always implicitly passed on your behalf. Okay? Do you know how, for like vector-nu, and vector-dispose, and vector-nth, vector-append, you always pass in the address of the relevant struct, as the explicit 0th parameter. Okay? Well, they just don't bother being explicit about it in C + + because if they know that you're using this type of – if you're using that type of notation, you're really dealing with object orientation. What really happens, on your behalf, is that it basically calls the minky function, not with one variable, but with two. It actually the address of something that has to be a binky class, or a binky struct, as the 0th argument. So whenever you see this in C + +, what's really happening is that it's doing this: it's calling the minky function, passing in the ampersand of b, and the ampersand of n. Okay? That happens to be an elaborately named function. And I'm just going with the name spacing to make it clear that minky is defined inside binky. Okay? And I'm writing it this way because even though we don't call it using that – that we don't use it – call it using this structure, right here, this is precisely how it's called at the assembly code level. Okay?

There's certainly an address of the first assembly code instruction that has anything to do with the allocation of this w variable. There's going to be an s p is equal to s p minus 4, at the top of the assembly code that gets admitted on behalf of this. People believe that, I'm assuming? Yes, no? Okay? The reason that C + + compilers can just be augmented, at least the code admission port of it, can be augmented to really accommodate some parts of C + + fairly easily, references and traditional method calls against an object, is that, whenever it see this, it says, "Okay, I know they mean this because they're being all LL about it. But they're really calling a block of code, okay, associated with the minky function inside the minky class, and I have to not only pass in ampersand of m as an explicit argument, but before that I have to splice in the address of the receiving object." Does that make sense? Okay. So the activation record that exists on behalf of any normal method inside a class, it always has one more parameter than you think But it still is gonna have a safe p c – I'll write it right there – it's gonna have two parameters, on top of it. This right there, this would be the thing that is locally referred to as z. Okay? Below that, it would make space for this variable uptight int called w. This right here might point to something like that; it certainly would point to something like that in this scenario, right here. Does that make sense to people?

Because n – I'm not using pure stack frame picture here – but because n is declared with a 17 right there, this would obviously be declared and initialized that way. Okay? Make sense? This would point to some assembly kind code instruction associated with after that call, right there. So there's a little bit more to keep track of, but as long as you just understand that k argument methods – k just being some number – k argument methods really are just like k plus 1 argument functions. Okay? When we're thinking in terms of a paradigm, we don't actually wonder about how C and C + + are similar. But when we have to write assembly code for something, we say, "Okay, I want to use the same code admission scheme for both C structs, with functions, and C + + objects, with methods." You can use exactly the same scheme, the same memory model, for structs and classes, and function call, and function call and return, by just looking at k algorithmic methods, as if they're k plus 1 arguments functions, knowing that the 0th argument always becomes the address of the relevant object. Okay? Does that make sense to people? When this, ultimately, calls this function right here, you understand that it's really equivalent to that. We just don't bother putting in the this pointer. Does that make sense? So internally, when I set up function call, or assembly code to actually jump to the binky colon colon donkey method, I actually have to make space for 12 bytes of parameters, 8 bytes for two copies of winky. Make sense? And also, the this pointer. And because there's nothing in front of this method call, it just knows to propagate whatever value this is, and replicate it down here for the second method call that comes within the first one. Does that sit well with everybody? Okay?

Had I had a variable of type binky reference here, then – and I had done, like let's say I had done this – I can't change the picture, but I'm just improvising this one point. If I had done something like this: binky of d – binky reference d, and done this, then the address of whatever binky object d refers to would have to be the thing that's laid down in the this portion of the activation portion of the record for the donkey method. Okay? That make sense to every body? Yes, no. Got a nod; it doesn't tell me. Okay. So even though

you think you're data-centric when you program in C + +, and you're verb function or procedure-centric, when you programming in C, the compilers really see them as just different syntaxes for exactly the same thing. Okay? And they ultimately become this stream of assembly code instructions that just get executed in sequence, with occasional jumps and returns back. And it just promises because the compiler makes sure that it can meet the promise. It just promises to emulate what the C – procedural C – or the object-oriented C + + programmer intended to do. Okay? Make sense? Yeah.

Student:[Inaudible] use stasis?

Instructor (Jerry Cain):That's completely different. We don't see static methods too much in 106 and 107. You know how the word static seems to have, like, 75 different meanings? Well, it has a 76th meaning, whenever the word static decorates the definition of a method inside a class. Okay? I can go over this. Suppose I have a class called fraction, okay, and publicly, I have a constructor fraction where I pass in an int m and an int denom and I might even default the denominator to be 1. So you can actually initialize fractions to pure integers. And I have this whole stream of methods. But then, I also have this one method inside, called reduce, which is intended to take a fraction that's stored as 8 4ths and bring it down to 2 over 1. Does that make sense to people?

Well, as part of that, typically what happens is that you'll write some function: int greatest common divisor int x and int y, and they're still put inside the fraction class because it's seen – it might only be relevant to your fraction class. And you actually pass in these two parameters and it really just behaves as a function because it doesn't need the this pointer, in order to compute the greatest divisor that goes into x and y. So a lot of times, what you'll do is you'll mark it as static. And what that means in the context of a C + + class, is that you don't need an instance of the class to actually invoke it. You can actually just invoke it as a stand-alone function. In fact, it really is a regular function that happens to be scoped inside the definition of a class as if it's just – as if the class has a name space around it. This is – I didn't mean to put public here, I meant to put private. You remember how on – this is how similar static methods and functions really are. Remember Assignment 2? Some of you had that headache of trying to pass in a method pointer to be searched, when it actually had to be a function pointer, and you're all like, "What's the difference. They're all the same to me." Static methods because they don't have this invisible this pointer being passed around, they really are internally recognized as regular functions. So if I wanted to define my act or compare function, not as a function but as a static method inside, I could have done that. I could have passed the address of a static function to b search. Does that make sense? Static method, I meant. Okay? Good? Okay. So static, I don't want to say you should avoid it. There are certainly places where static is usually a good thing to do. It interferes with your ability to use inheritance when you're dealing with C and C + +. You don't get inheritance and you don't get – you don't get the right thing, in terms of inheritance, when you're dealing with static methods. Which is why you don't see them as often. And normally, when things are written as methods inside a class like this, it's because they are actions against objects, as opposed to actions on behalf of the class, like this would be, right here. Okay? Any other questions? Okay. So next Tuesday, when we pick up on the normal discussion

section cycle, you will get a section handout, where you'll have some elaborate – I don't want to say elaborate – some meaty examples on C and C + + code generation. I'm only gonna test you on C code generation, with no objects, and no references on the mid-term, although it'll be fair game on the final.

Next Wednesday, I will give out Assignment 5; it will not be a programming assignment. It'll be a written problem set, where you're not required to hand it in. You're just required to do it, and make sure that your answers sync up with the answers that I give out in the key. And it'll be totally testable material. You're not – you're just required to effectively do it by 6:59pm, on Wednesday, May 7th. Okay? Does that make sense? So you have a full week to do this one assignment, which means you'll all do it on Tuesday, May 6th. But you'll have a week to actually kind of recover from the Assignment 4 experience, and learn all the pointer stuff. Okay? This make sense? What I want to do is I want to start talking, a little bit, about how compilation and linking works. We completely disguise compilation and linking with this one word called make. And it's just like – it's this magic verb like, "Just do it, and make it work for me." And somehow, it actually, usually does do that for us. But when you compile C code – C + + codes as well – but C Code, I'll focus on. It normally invokes the preprocessor, which is dedicated to dealing with pound define and pound includes. Then it invokes what is truly called the compiler. That's what's responsible for taking your dot C files, and your dot C C files, and generating all those dot o files, that happen to appear in your directory, after you type make. Make sense? And then, after compilation, there's this one thing you don't think about because code warrior, and x code, and visual studio C + +, they all make compilation look like it's the full effort to create an executable. But once all those dot o files are created, and one of the dot o files has code for the main function inside, there is this step called linking, which is responsible for taking a list of dot o files; stacking them on top of one another; making sure that any function call that could possibly ever happen during execution, is accounted for because there's code for that function one of the dot o files.

And then, it just wraps a dot out, or 6 degrees, or my executable, or whatever you want to call it, around the bundle of all of those dot o files. Does that make sense? Okay. Somebody had mentioned, the other day, that a line like this, why wouldn't it be replaced by call of, like, p c plus 480, or p c minus 1760? To actually jump to the actual p c relative address where the swap function starts, that actually will happen in response to the linking phase because everything's supposed to be available to build the executable file. And so if it knows where everything is, in every single dot o file, and it knows how they're being stacked up one on top of one another, they can replace things like this, with their p c relative addresses. Maybe, at the time that things are linked together, it knows that swap function happens to be the one defined rate above this. So when it calls swap, it actually calls p c minus , let's say 96, or something like that. Okay? Make sense? Okay. Let me get some clean board space, and just spend five minutes, that's all I'll be able to do, really, talking about the preprocessor.

And I'm doing – okay, five minutes. Whenever you're dealing with pure C, if you wanted to declare global constants, traditionally until recently, you only had pound defines as the

option for defining nice, meaningful top-level names for these magic numbers, or these magic strings. So something like this: pound define – I'll just do k with, let's say it's 480 pixels, so I write down 480. And then I have something like this, k height 720. And then, I have some pound includes; I'll talk about those later. And then, there's all this code right here, where usually, it's the case that some part of it is framed in terms of that and/or that. So maybe it's the case that you have a printout statement with is percent d backslash n –oops, right there – and then, you feed it the value of k width. Maybe a more meaningful thing might be something like int area is equal to k width times k height. And it exists in your code somewhere. When you feed, or when a dot C file is fed to GCC or G + +, there's a component of either GCC or G + + called the preprocessor that doesn't do very much in terms of compilation. It doesn't do compilation, as you know it. It actually reads your – the dot C file that's fed to it, from top to bottom, left to right, like we would read a page. And every time it sees a pound define, it really only pays attention to these words. What it'll normally do is, it'll read in a line. And if there's nothing that has a hash at the beginning of the line, so it doesn't involve any pound defines or pound includes, it'll just say, "Okay, I just read this one line, and I don't care about it. So I'm just gonna publish it again as part of my output." When it reads something like this, it internally has something like a hash set, although it's probably – I don't want to say it's strong, eh, but it's probably strongly typed.

It associates this as a key with this string of text; it doesn't even really know that it's a number. Okay? It just says, "Everything between the white space that comes right after width, and the white space that comes after the first token, that's associated as a text stream with that identifier, right there." Okay? It does the same thing for that, right there, and as the preprocessor continues to read, every time it finds either that or that, in your code – the one exception being within a string constant – but everywhere else, it says, "Okay. Well, when I see that, they really meant that number right there." So it lays down 480, as if you typed it there explicitly. Does that make sense? Okay? You know the alternative would be to put lots of 480s all over the code base, without using a pound define, or lots of 720s around the code base, without using a pound define. But then you change the dimensions of your rectangle, in your graphics program, or whatever. And then, you have to go through and search and replace all the 480s, to make them 520s or whatever. It makes some sense to consolidate that 480 to a pound define, if the pound define is the only thing that is available to you. But the pound define is really very little more than pure token, search, and replace. Okay? Does that make sense to everybody? So the responsibility of the preprocessor is to read into the dot C file, and to actually output the same dot C file, where all the pound defines and other things, like pound includes, have been stripped out. And the side effects that come with pound defines have been implanted into the output. So this, right here, would be missing these two lines. Anything right here, that doesn't involve those two pound defines, would be output verbatim. This one line would be output verbatim, except that would have a 480 replaced, right there. Okay? And this, right here, would have [inaudible], and that right there, it would probably have a 480 times 720. Although, the compiler might actually do the multiplication for you, if it's a very good compiler. Okay? Does that make sense? Actually, the – I'm sorry – the preprocessor would not do that, though. This would just become 480 times 720. Okay? And the fact that it happens to be text – it is really text, at

the moment – it's just that, when it's fed to the next phase of compilation, it'll go ahead, and it'll chew on those things, and recognize that they really are numbers. And that's when it might do the multiplication. Yeah?

Student:[Inaudible] define k width and then you declare a variable called k width underscore second, or whatever?

Instructor (Jerry Cain):Yeah, it won't do that. It has to be – it has to be recognized as a stand-alone token. So it I did this, for instance, like k width underscore 2, something like that, it'll not do sub – sub token replacement for you. It also won't do it within string constants, either. Okay? Okay. I'm not done with the preprocessor yet, but I can certainly answer your question. Go ahead.

Student:[Inaudible] verbatim? [Inaudible] printouts?

Instructor (Jerry Cain):Yeah.

Student: What does the preprocessor do with that line?

Instructor (Jerry Cain):The preprocessor would, basically, output this line right here, except that the k width would have been replaced – spliced out and replaced with the 480. Okay? Which would be – it's supposed to be functionally equivalent. It's just allowing for the consolidation of magic numbers and constants. So whatever we want to use the pound defines for, to just consolidate all of this information at the top. Okay? Yep?

Student:[Inaudible] preprocessor [inaudible]

Instructor (Jerry Cain):It actually does not know the 480's a number yet. It is – it is – they just are incidentally digit characters, and that's it. But if I had done this, like that right there, the preprocessor would be, like, "Okay. I'm just gonna take that 6 character string, and put it right there, and right there." And it's only later on, during real compilation, post preprocessor, that it'll be, like, "Hey, you can't do that." Okay? Does that make sense? Okay. And that wouldn't happen. Okay. So come Monday, I will finish up the preprocessor. Pound defines are easy; pound includes are not. They're actually some involved with that. And then, I'll talk about compilation and linking. Okay. I will see you on

[End of Audio]

Duration: 51 minutes