ProgrammingParadigms-Lecture01

Instructor (Jerry Cain):For 240, now there won't be.

Student:I can sit in [inaudible].

Instructor (Jerry Cain):Oh, that's fine. Hey, everyone, welcome, we are live on television. So this is cs107. My name is Jerry Cain. I have two handouts for you today. They're the type of handouts that you expect to get on the first day of a course. As far as pertinent information about myself; my life synopsis is on the board right here. There's my email address, those are my office hours, and that is my office upstairs, just more or less, directly above this room. By far, the best way to get in touch with me is email via that address. Certainly if you have a question that you need to ask me specifically about then email is great, I'm very good about checking email. I'm the type of person that just clicks on send and receive every three seconds, so I know when your email comes in so I'm always very good about responding. The office hours are early in the mornings on Mondays and Wednesday, I know that's not the best time in the world, but just for various reasons, that's the only time I can really have them, but in general, if I'm in my office you are more than welcome to stop by and you're always welcome to call my office to see if I'm in during some other time in the week. Okay. I'll try to make it a point to just casually be around the office as much as possible, but Monday and Wednesday, 9:00 to 10:30, is really the official time when I'm around. Okay. I have a good number of staff members; I can have them just wave, you don't need to stand up, just wave, you can wave a little bit more enthusiastically, thank you. Okay. That actually only represents about a third of what the staff will be. Some people have a conflict today so they're not here. And I actually only figured out last night how large this class is. We expected a 130 - 140 students and I just checked at 10:55 that we have 241 signed up for the course, which is the largest that cs107 has ever been. It's really weird having this many people come and take and interest and listen when you talk.

I feel a little bit like Barack Obama. I'm delighted that this many people are taking it and so I certainly expect to have, on the order of 15 to 18 TAs and undergraduate section leaders working and they're very pertinent to the course. I almost require – in fact, it is the case this quarter, that every single person I'm gonna hire has taken 107 either as a undergraduate or as a first year graduate student, so they are great resources for questions and their office hours, with very little exception, they've all done all the various assignments that you'll be doing this quarter, so they already know what your questions are gonna be. Okay. As far as their office hours are concerned, they will have office hours and they'll rotate through a grid of evening hours at the main computer cluster where you'll be expected, or at least invited, to do all of your online programming work. Okay. Those will start next week. You're not gonna have an assignment until this Friday, and I make that one as easy as possible as far as the coding assignment is concerned because you have to get used to Unix and this new development environment. A lot of you have probably never been there before, so I go soft on your on assignment one so you can actually not be intimidated by the whole Unix development process. We do have a discussion section. We had scheduled one for tomorrow – I'm not even gonna say the time because I don't want to confuse people – but it turned out that it was terrible for three of my four head TAs so I'm in the process of rescheduling that. It's not gonna be on Thursday, it's probably gonna be on Tuesdays. That's what I'm shooting for now. At the time I had to take these handouts to press I just didn't have a time yet, but I will announce, certainly in lecture on Friday and probably prior to that via the emailing list. You're not required to go to discussion session, it is gonna be televised just like the lectures are – there will always be section handouts and section solutions, so even if you just don't want to bother watching it, not that I'm recommending that, but you certainly have the resources to figure out what they went over during that section because I'll be very good about putting up section handouts and section solutions so you know exactly what topics they were discussing. Okay. Whatever the time is next week, it'll probably be Tuesday sometime in the afternoon.

The first section, if you go to one section off quarter and go to one live section off quarter, next weeks is probably the one to go to if you've never really dealt with Unix before. They'll invest some energy coming up with some slides and some examples to show you what it's like to open an [inaudible] in Emax and to actually code there and then to use a command line to compile the program and see how you actually compile and execute and debug programs in a Unix world. Okay. And it's a little weird at first, you kind of hate it for three or four days then you start to like it because it's really light weight and very fast. So that's another reason why I make the first assignment less about coding and more about just getting used to Unix. Okay. Not next week, but the week after when we start having discussion sessions again, and having more of them, that'll adopt a more traditional format where we bring section problems and we talk and everybody raises their hand and asks questions and things like that. Okay. As far as reading material is concerned, in the handout I specify three books that I think are really great books. Two of them are on C++; they're actually quite advanced books. There's also one book on a language called Scheme, which I'll talk about in a second. These are by no means required and I don't even want to say that I recommend them in a sense that those who are really fastidious about going and buying every book they think is gonna help them, I'm not really even saying that you should go buy it, I just want them to be on your radar so that if after you get through the C and C+ segment of the course, you can see yourself doing more serious C++ programming later on, then you might want to go and get these book. But don't spend money on books yet because this class has more or less been taught with the same type of structure over the past 15 years so we've compiled pages and pages of handouts. This is just the tip of the iceberg. Anything you're gonna be responsible for is covered either in lecture of in the handouts. I'm not gonna have any external reading from text books so just worry about the handouts. The handouts are very polished. I think they're in really good shape. You'll average one to two handouts every single lecture and probably Friday and Monday, you're gonna get four or five each day. As far as the exams are concerned – let me talk a little bit about that. Midterm, there is one, and it is Wednesday, May 7. Okay. I schedule it for 7:00 to 10:00 P.M. People roll their eyes when they see a three-hour window set aside for a midterm.

I'm not gonna write the midterm with the intent of you needing every single minute of those three hours. I genuinely make an effort to make the exam so that it can be completed, patiently, in 60 to 90 minutes, okay, and occasionally, I'm off. When that happens, you have this buffer of an extra hour and a half to really work through it. Okay. I just give people three hours with the intent of giving the illusion of an infinite amount of time so that they don't feel that they're pressured to get through it. That may sound like I'm being really nice, but I'm actually being slight cruel by giving you three hours because if you do badly on it, then you have one less reason to blame me. You actually have to work through the problems and you really have to know the material, and you can't blame time pressure as being a factor as to why the exam didn't go well. So that's why I want to give people as much time as I can reasonably give them without going into 2:00 A.M., 3:00 A.M. Okay. I recognize that because this is scheduled outside of lecture that many of you may have classes, orchestra meets at night, people have various activities they have to do at night, I want to displace you from those because I decide to have my exam outside of lecture so if you can't make that time, that's fine, just let me know soon and plan on taking the exam during some three hour window that fits between 10:00 A.M. and 6:00 P.M. Okay. So most people that works with. If that's not gonna work out, then let me know now. I'll make sure you take the exam sometime, but I'll be around more or less or TAs will be more or less all day Wednesday to make sure that you can take an exam when you need to. I'll accommodate any reasonable requests. As far as the final is concerned, I offer the final exam twice. Our dedicated slot is Monday, June 9 and the official time is 8:30 A.M. Now, because we're on TV and because Stanford students – there's just not that many times to offer classes, a lot of people have conflicts, a lot of people are taking two classes, sometimes even three classes Monday, Wednesday, Friday at 11:00, and if that's the case, then you have two or three finals on Monday, June 9th at 8:30. If you want to take all three then, you're more than welcome to, but I assume you're not gonna want to do that so if you can't make this time, it is fine, provided you can make an alternate slot of 3:30 P.M. that day. If you cannot make either of those times, please let me know now so I can take care of that.

As far as the breakdown of assignments versus exams, this is how it all pieces together in the end. You have assignments, you have the midterm, you have the final, these contribute 40 percent of your grade. The midterm is the least of the three and the rest is the final. Okay. That's a pretty sizable chunk is dedicated to the assignments. People, in general, do very well in the assignments. They might struggle to figure out what kind of things we're expecting on assignment one, but eventually, people figure out what I'm expecting and they get the programs to work and they write nice clean code, so we see a lot of A's and A-s on the programs. We just do. And I imagine 80 percent of you will be getting A's and A-s on some of the later programs because you'll just start figuring things out. And that's not to say that grades will be much lower prior to that, but just toward the end everybody seems to be doing very well on the assignments. There's probably gonna be six or seven assignments. One of them will not be handed in, it'll be a written problem set where the deadline will, more or less, be at the time you take the midterm because it'll be a series of written problems that serve as practice for the midterm. You're certainly responsible for that, but all the other assignments take a programming assignment approach where you actually code up stuff and you electronically submit it and one of the section leaders or the TAs grade your work and they electronically or they email you back feedback. I don't need any paper copies at all. Everything is dealt with electronically. We're gonna really put a lot of effort into trying to turn the assignments around very, very quickly. In order to do that, I do allow you to turn work in late. I have the same type of late day policy that 106b does.

I actually give you the ability to grant yourself little extensions in 24-hour units. I give you five of those little days. And if you want, you can use one or two or all five later days on any particular assignment, but after five days, whether using free late days or extra ones that actually cost you a little bit, I need all the assignments within five days of the deadline so that I know that the TAs can start grading them and crank out grades and crank out feedback and get them back to you. Does that all make sense to everybody? Okay. As far as the midterm or final will go I actually try to make this – I don't want to say that I make it easy, I think, of the two exams, the final is a little bit more difficult because I think the midterm actually deals with the more difficult material so I go soft on you here knowing that I'm gonna revisit some of that material on the final. Okay. Now, when I say, "Soft," I don't mean easy, I just meant softer, comparative. Okay. Then I would normally go, but nonetheless, that's why I have the breakdown of 25 to 35 – really, all the material is equally represented on the final exam. I'm trying to think what else is going on. Oh, of course, as far as resources for keeping in touch with us, if you have a question specifically for me, then by all means email that address right there, but if you're asking a question about an assignment or lecture material that you know or suspect any section leader or TA would be able to answer, then you're gonna benefit by sending mail to this address. Cs107@cs.stanford.edu. That's precisely the address that you would assume would be attached to a staff emailing list for this class. When you do that, you're sending mail to an account that all 15 or 18 or 19 of us whole on a fairly regular basis and so you're that much more likely to get a quick response. Maybe not so much on a Saturday morning, but certainly on a weeknight when everybody's coding and deadlines are approaching. We're very good about pulling that and getting responses out as quickly as possible. Okay. If you don't hear back from this within, like say, 12 or 24 hours – let's say 12 hours in a weekday and 24 hours on the weekend, then it's okay to email me directly and say, "Hey, I'm not getting my answers, can you answer this question for me?" But be patient with that. We're actually very good – very often, we get back to people within minutes. A lot of times the TAs are sitting in front of the computer right next to you in the term or cluster where you're working, you just don't know what they look like so they respond to your question very, very quickly. I'm also gonna try to experiment – I have no idea what I'm planning to do with this, but I have a Facebook group. The URL is for cs107 – the URL is in the handout. You do have to be a member of Facebook.

I'm not trying to evangelist Facebook just because I work there as well, but if you have a Facebook account already and want to go join this Facebook group, just visit the URL that's posted in the handout or just do a search for cs107 and I'm sure it'll come up. I also have a Twitter account for 107. If you want to follow cs107 – I think this is an interesting idea because I could send little announcements, these very lightweight, noninvasive announcements to everybody who's paying attention to cs107 on Twitter saying, "Webpage has been update or exam cancelled," or something like that. Okay. And you can all follow what's going on there. I don't want you to think that you have to pay attention to these. Anything that's truly important about assignment deadlines or things that need clarification, I will actually use the mailing list and post to it. That will be the official forum for things that really matter but these are intended to be lighthearted. As far as that mailing list is concerned, I have 241 people signed up for the course already. That means that most of you, if not all of you, are automatically enrolled in the mailing list for the course, so when I send mail to cs107-spr0708-students@list.stanford.edu, you automatically get that email provided you're registered for the class. If you haven't registered for the class yet, let me pester you a little bit to go and sign up for the class even if you're not sure if you're gonna take it because, probably, I send more announcements in the first week of the course than I do anytime later on because just a lot has to happen and we have to up ramp a lot in the first few days so I'm certainly gonna send an email out as soon as I found when section time is. A lot of you have signed up for the course may not realize this, but sometimes you can inadvertently set some privacy settings so that you're email is not put on that list – do you know what I'm talking about? Okay. So go to Stanfordu.com or u.stanford.com, I forget what it is or wdu where you just go make sure that your email is public just to the Stanford community so that you can get on that emailing list. If you have a problem with that, I understand that you don't want your email address advertised, but then just make it a point to email me and say that you're not that so I can put your email address on some sub list that I always include with the announcements. Does that make sense to everybody? Let me just talk a little bit about the syllabus. Let's make sure there's nothing in here. Is there anyone in the room who has not taken 106b or 106x, a couple people I'm assuming, okay. I'm just taking a pulse on this. I don't necessarily require 106x or 106b, specifically, but I do kind of expect that you've done some C++ programming, that the notion of a linked list or a hash table is familiar to you, a binary search tree, a function point or all of those things that are covered in our 106 courses are familiar to you. If that isn't the case or you don't know C++ all that well, then I'm a little worried because the first half of 107 is all about advanced C and C++, so if you don't have the basics down or you haven't been exposed to that stuff yet, it makes the first two weeks of the course, which is normally actually quite fun because you're learning all about the inside, under the hood machinery of the C language, it can make it pretty miserable unless you actually anticipate spending a little bit more time getting up and running. So if you don't know C or C++ and you've never written an assignment for a course in C or C++ then talk to me after lecture so I can kind of gauge as to whether 106b or 107 is better for you. Okay.

Here is, in a nutshell, a reduced version of what's presented in handout two. This is the syllabus. I have several languages that I'm gonna put on the board and one concept. C, Assembly, C++, Concurrent programming – that's not a language, that's just a paradigm, Scheme and then it is official, I'm actually gonna cover Python from now on in 107. People look at this and they go, "Wow, I'm gonna be able to legitimately put all of these languages on my resume," and it's from a separate list at the bottom and it feels really good. It isn't so much about that. Certainly we want to give you some mileage in some very relevant languages that are very good for both research and for industry, but the real intellectual value in learning all of these languages is to really study the paradigms that they represent, and I'll explain what that means in a second. I think a lot of C++ programmers really program in C and just incidentally use the objects within the classes that are available to them, okay, which is a perfectly reasonable way to program. Most people learned how to program in C, at least the people I know in the industry, know C very, very well, and in spite of the fact that there's 50 million newer languages that are better in many regards, they still stick with what they know and that's why C and C++ are still such popular languages. There's nothing wrong with programming in C if you know it very well and you write clean, readable code, it's just more difficult. I don't really care so much about teaching you how to program an assembly, I use it as a vehicle for showing you how C and C++ programs compile to dot 0 and to object files and to binaries and that become executables and show you how a line like I = seven or J++ or FU of X and Y is a function call, how that all translates to assembly code. Okay. Does that make sense? You know when you write C++ code that when you execute the program, it's not C++ anymore, it's assembly code. It's all zeros and ones eventually. I want to give you some insight as to how C is translated to assembly code, how all these variables and your functions and your objects and all of that, eventually, get boiled down and hashed down to a collection of zeros and ones and I want to do a little bit of the same thing for C++. It turns out that – well, C++ and C represent different paradigms, that they really compile the zeros and one, and after you get enough experience with this assembly and the manual compilation process that we're gonna learn about and how to look at C code and figure out what the assembly code would look like, you're really gonna see that C++ and C almost look like the same language as far as the zeros and ones are concerned. I'm gonna be able to do something like **&**P arrow ***=7 and you're gonna know exactly what it means. Okay.

So it takes a little bit of work and it's almost laughable how arbitrary you can be with formulas, but if it compiles, it means something so when it runs it actually does something. It's probably not good if you have a lot of asterisks and ampersands, but nonetheless, you can have some idea as to why it's crushing, not just that it is crushing. Okay. I spend a good amount of energy talking about concurrent programming. We actually – at the moment, do that type of programming in C, but all the programs you've written in the past two quarters, if you've just taken the 106A and 106b courses here or 106x, all the programs you've written at Stanford, prior to 107, have been sequential programs. That means that whether it's object oriented or procedurally oriented, you have this outline of steps that happen one after another. Nothing is done in parallel or pipelined or done side-by-side, everything happens in one clean stream of instructions. Okay. Well, what concurrent programming is about is within a single program trying to get two functions to seemingly run simultaneously. If you can get two functions to seemingly run simultaneously then you can extend that and get 10 functions to run simultaneously or 20 functions to run simultaneously or seemingly simultaneously. I say it that because, technically, they don't run at the same time. When I go over assembly code, and I think you can intuit enough about what assembly code is like, but if you have this one function, okay, my hand is a function, this hand is another function, okay and you concern yourself with the execution of one of them and then when I do this you can just think about it reading the code and executing it for side effects. Does that make sense to people? When you deal with concurrent programming you have two or more functions, just two right here because I only have two hands, to do this and they both seemingly run at the same time, but what really happens is it's like watching two movies at the same time where –

because there's only one processor on those machines, it doesn't really run like this, it runs like this and switches back and forth between the two functions, but it happens so fast that you just can't see the difference. Okay. It's more than 24 transfer seconds, it's, like, a million. There are a lot of situations where concurrent programming is not very useful, but there are several situations, particularly networking, whenever that's involved, where concurrent programming is actually very useful.

There are some problems that come up when you deal with concurrent programming that you might not think about. The example I always go over the first day of class is just that it uses two Wells Fargo ATM machines. Okay. Think about you have a Wells Fargo checking account, you may not have to think about it because you probably do, so just imagine your checking account is in danger because two people are using ATM machines and you have a $100 in it and you share your PIN with your best friend and you go up to a neighboring ATM machines and you make as much progress as possible to withdraw that $100 and then you both, on the count of three, press okay to try and get $200 collected. Does that make sense? That is not a sense sensible example because both of those machines are basically very simple computers, okay, that ultimately need to access the same master account balance to confirm that $100 is available and in this transactional – transactional makes sense both in terms of money and also in the sense of concurrent programming – you have to make sure that the $100, being identified as the account balance, is maintained with some atomic flavor so that if you have two people trying to withdraw $100 at the same time that only one person gets away with it. That $100 account balance is the shared resource that two different processes have access to. Does that make sense to people? Okay. So there has to be directives that are put in place to make sure that the account balance check and the withdrawal are basically done, either not at all, or in full so that it really does function, truly, as a transaction, both in the finance and the concurrent programming sense. Okay. As far as this Scheme and Python are concerned, once we get through concurrent programming we really switch gears and we start looking at this language called Scheme. You may not have heard of this. If you haven't heard of this, you may have heard of a language called LISP, which it's certainly related to. This is a representative of what is called the functional paradigm. Okay. There's two things about Scheme and functional languages – purely functional languages that are interesting in contrast to C and C++. When you program using the functional paradigm you always rely on the return value of a function to move forward. Okay. And you program without side effects. Now, that's a very weird thing to hear as an expression when you've only coded for a year, but when you code in C and C++, it's very often all about side effects.

The return value doesn't always tell you very much. It's a number or it's a bouillon, but when you pass in a data structure by reference to a function and you update it so that when the function returns, the original data structure has been changed, right, does that make sense? That's programming by side effects. Well, the idea with Scheme and particularly the functional paradigm is that you don't program with side effects. You don't have that at all. You always synthesize the results or partial results that become larger partial results that eventually become the result that you're interested in and only then are you allowed to print it to the screen to see what the answer to the problem was.

Okay. It's very difficult to explain Scheme if you've never seen it before in a five minute segment of a full introduction, but when you get there, we have tons of examples of the paradigm. It's a very fun, neat little language to work in. This language called Python – I'm suspecting that most people have heard of it even if they've never seen it. It seems to be the rage language at a lot of significant companies in the Bay Area. They're very smart people with these companies so when they use a language and they like it, there's usually a very good reason for them liking it. You've probably heard of a language called Pearl. Okay. It's not a very pretty language. You can just think about, in some sense, Python being a more modern, object oriented version of Pearl. Okay. Now, if you don't know Pearl and you don't know Python it doesn't mean anything to you, but just understand that this sexy little language that's been around for probably 16, 17 years that's really established itself as a popular language since year 2000, 2001. I know a lot of people who work at Google that program in Python on a daily basis. There's a subset of us at Facebook that program in Python every day. It actually has a lot of good libraries for dealing with web programming. Web programming can seem really boring to a lot of people because it just seems like it's just HTML and web pages and things like that. Real web programming is more sophisticated than that. You dynamically generate web pages based on content and databases and things like that and Python, being a scripting language, which means its interpretive and you can type in stuff as you go, it recognizes and reads and executes the stuff as you type, it's very good for that type of thing, and if all goes well, meaning I have time to develop this assignment idea I have, you're gonna write a little miniature dynamic web server in Python for your final project. It won't be that sophisticated. You're not gonna write all of apache, but you are gonna probably write some little thing where you really do have a web server behind the scenes making decisions about how to construct an HTML page and serve that over to a client. So it'll be an opportunity to learn Python, to learn with libraries, to see that as a language because it's fairly young, it has the advantage of not bothering to include C and C++'s and Java's mistakes. It says, "No, I'll leave that part out and I'll go with this more interesting, well formed core," it has great libraries, it has object orientation, you can program procedurally if you want to and just program like you're in C, but using Python syntax.

There are even functional programming aspects in the language so even though the syntax is different from Scheme, conceptually, you can use Scheme like ideas in Python coding if you want to. Okay. I'll be able to illustrate the client server paradigm and how it's different from traditional programming. That's not so much a Python thing, but Python is a good vehicle for learning that stuff. There are a few other paradigms that aren't represented here, but I think I really cover all the ones that you're likely to see for the next 15 years and you're a coder. Okay. There are a couple of the languages that I may briefly mention the last night that are just fun, but they all have some overlap with some languages represented right here. Okay. You guys are good? Okay. So I don't like starting in on any real material when I only have 10 minutes left, so I'm actually gonna let you go, but recognize that Friday I'm gonna have tons of handouts for you, I'm gonna have an assignment, okay, we're gonna dive right into the low level pointer stuff of C and C++. Okay. So have a good week.

[End of Audio]

Duration: 37 minutes

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

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

ProgrammingParadigms-Lecture04

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[End of Audio]

Duration: 51 minutes

ProgrammingParadigms-Lecture05

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

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

So this is what I wrote last time.

void *lsearch

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

void *key

void *base int

m is the length of that array

int lm size is the size of the elements

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

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

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

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

void * lm (address)

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

This is equal to numerically:

base plus i times om size.

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

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

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

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

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

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

This is easy to understand. It's easy to just look at this and understand what's going on because you know what linear search is; that's not the hard part. The hard part is getting all the pointer math correct in the char * trick, and actually aligning these things up properly, and invoking the comparison function properly.

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

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

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

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

void * lm 1

void * lm 2

Just because some – let's write the seven right here, this is the thing called number. On an arbitrary iteration, it may pass this int as the first argument to the comparison function. This right here is being invoked right there. It's gonna pass in the address of that one isolated seven right there every single time. The second parameter's gonna get that, and if it fails then that, and if it fails then that, etc., until it runs out of space. I have to return a -1, or a +1, or a 0 depending on whether they match or not. I also, because I am writing this function, specifically to make this call, this constrains the prototype to take two void *'s, but I know that they're really int *'s. So because I'm writing that code as a client, I can reinterpret the void *'s to be the int *'s that they really are. So I will do this, int * ip 2, and I will just set it equal to lm 1 and lm 2. It turns out in a pure C compile you do not need to do a cast there, it just understands that the cast is implicit; it has to do it. So I have these local variable, ip 1 and ip 2, that not only point to this and that right there, but they actually understand them to be four-by quantities to be interpreted as integers. Does that make sense? So all I have to do is return *ip 1 - *ip 2. I'm relaxing a little bit on the return type, I'm letting zero meet a match. And of course, if that difference is zero than of the same number, -1 and +1, I could constrain it to be that. I just want it to really be a negative number or a positive number to reflect the delta between the two. Does that make sense to people? So if you understand this, great. If you understand this, even better. I'm sure most of you understand this even if it's the first time you've seen this type of code. Once we actually understand how all of this stuff works you're gonna be very, very happy. It's a little hard to understand the very first time you see it. But you have to recognize that this is not exactly the most elegant way to support generics, it's just the best that C, with it's specification that was more or less defined 35 years ago, can actually do. All the other languages you've ever heard of they are all so much younger that they've learned from C's mistakes and they have better solution for supporting generics. There are some plus's to this. It's very fast. You only use one copy of the code, ever, to do all of your lsearching. The template approach, it's more type safe. You get more information and compile time, but you get code bloat because you've got one instance of that lsearch algorithm for every single data type you ever searched for. Does that make sense? It’s easier to get this right because you’re not dealing with atomic types that are themselves pointers. We have integers right here. This gets a lot more complicated when you start dealing with the problem of lsearching an array of C-strings. Okay. So you’re going to have an array of char *’s, and you’re gonna have to search for a particular char * to see whether or not you have a match or not. These are the int boards. I want to deal with this setup right here. I have a small array 1, 2, 3, 4. And let’s say I have an array of C-strings. Let’s just assume that it’s initialized this way. And I have an array of five little notes there. And I want to search the array using lsearch for an E-flat. So here’s my key that I’m searching for; it points to an E-flat. I should emphasize that these are really character arrays that are null-terminated. Same thing with this. This right here is a character, character, character, character. This is a character array. That means that’s a char *; char *, char *, char *. The address of the array right there, the arrow I’ve just drawn in, is technically of type char * *. How is lsearch gonna absorb that? It’s going to absorb it as a void *. The only way it’s gonna be able to compute that address, and that address, and that address as part of the linear search is because we’re also gonna communicate the size of a char *, so it can manually compute the addresses of all those boundaries. The comparison function that needs to be written in order to do this has to be willing to accept addresses of that type and that type right there. This is where things can get confusing because you can kinda drift back and say, well, everything’s a pointer so why does it matter that I pass in this as opposed to this? You’re gonna see, when we write the comparison function, that the number of hops from the tail of the arrow that’s passed in really matters. If lsearch passes this type of pointer to your comparison function then you really are two hops away from the actual characters that are compared to one another. Does that make sense to people? You may ask, well, why doesn’t the comparison function just pass these pointers in? The answer is that lsearch has no idea that those things are really pointers. The only thing it knows is if they happen to be for four-by-fourth of information. Make sense? Let me declare this: char * notes array is equal to, I'll write them as string constants, A-flat, F-sharp, B, then G-flat, and then an isolated D. I can talk about how these things are stored in a little bit. They're not in the heap, they're actually global variables that happen to be constant. It's like normal global variables, except they happen to be character arrays that reside up there, and these are replaced at load time with the base addresses of the A, F and the D.

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

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

char * * (found)

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

Student:

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

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

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

Student:[Inaudible?]

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

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

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

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

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

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

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

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

Student:[Inaudible] char *?

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

Student: Yeah.

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

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

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

Student: Yeah.

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

Student: Actually, I understand that [inaudible].

Instructor (Jerry Cain):This right here?

Student:[Inaudible.]

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

Student: Yeah.

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

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

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

Student: Just like referencing the [inaudible].

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

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

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

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

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

Student:[Inaudible.]

Instructor (Jerry Cain):Well, I can say the way I wrote it first is the way it's typically written. Because of the symmetry, I think coders, I don't know if they like to see it, I think they're just in the habit of only dealing with comparison functions that really deal with the same incoming data type. And that's not the case if one's a char * for real and one's a char * * for real.

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

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

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

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

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

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

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

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

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

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

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

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

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

Instructor (Jerry Cain):This right here?

Student: Yeah.

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

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

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

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

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

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

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

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

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

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

And then I want to do the following:

int * lm's

int logical (length)

int allocative (length)

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

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

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

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

void * stack (new)

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

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

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

Void stack push

What stack am I pushing onto? The one identified by address right there. What integer's getting pressed? This one. And actually we'll go with an int right here, stack pop. Which stack am I popping off of? The one that's identified by address right there. I just want to be concerned with those things right here.

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

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

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

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

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

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

I do this [inaudible] int i = 0; i < 5, i++. I want to go ahead and I want to do a stack push. Which stack? The one at that address, and I just want to pass in i. Just draw the picture as to how everything's updated. And then right here, rather than dealing with the pop problem, which is actually not anymore difficult than the push problem, I just want to go ahead and stack dispose & of s.

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

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

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

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

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

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

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

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

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

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

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

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

[End of Audio]

Duration: 52 minutes

ProgrammingParadigms-Lecture06

Instructor (Jerry Cain):Hey, everyone. Welcome. I have one very short handout for you today. It is the handout that has two problems that we'll be going over during tomorrow afternoon's discussion section. Remember, we don't have it at 3:15; that was just last week to accommodate what I assumed would be a large audience. It's at 4:15. It's actually permanently in the room down the hall, in Gates B03. So I'm not teaching the section, Samaya is, who I think is here, who is handing out the handout. So look for that guy tomorrow at 4:15.

Both of these are all the exam problems, so they're certainly good problems to understand the answers to. And if you're not gonna watch the section or attend, just make sure you at least read the handouts. You may very well be able to do the problems yourself, and if so then that's fine, but if not, make sure you watch the discussion section at some point.

When I left you on Friday, I was a quarter way, a third a way through the implementation of an int-specific stack. So there's nothing technically cs107 level about this, except for the fact that we're being careful to write it in pure C, as opposed to C++.

Now, if you remember the details from Friday, about the interface file, the functions are, I mean, I'll talk about those in a second. We actually expose the full struct. There are no classes, and there's no public, and no private. So everything is implicitly public in the dot age. That's a little weird. C++, when you learned about classes and objects, it was all about encapsulation and privacy of whatever could be private. We can't technically do that in C; although, we can certainly write tons and tons of documentation saying just pay attention to the type, use these functions to manipulate it, and pretend that these are invisible to you.

You more or less operate as if this is a constructor, a destructor, and methods, but they happen to come in pure, top-level function form, where you pass in the structure being manipulated. So StackNew is supposed to take the struct, one of these things, addressed by s, and take it from a 12-byte block of question marks to be logically representing a stack of depth zero. This is supposed to kill a stack at that address. This is supposed to increase its depth. This is supposed to pop something off.

I wrote StackNew and StackDispose. I certainly wrote StackNew, I'm forgetting about StackDispose, but I'll write them really quickly right now.

This is the dot c file:

Stack .c

void StackNew

Stack

I add these three things:

Stack logLength = 0 (that's because the stack is empty)

I'm going to make space for four elements, and then I'm going to go ahead and allocate space for four elements using c as a raw dynamic memory allocator, called malloc. It doesn't take anything related to a data type. It doesn't know that anything related to a data type is coming in as an argument. You have to pass in the wrong number of bytes that are needed for your four integers. And that's how you do this. All that malloc feels is an incoming 16. It goes to the heap and finds a figure that's that big, puts a little halo around it saying it's in use, and then returns the base address of it.

I told you to get in the habit of using a certain macro, that was mentioned a little bit in Assignment 1, just to confirm that elems ? null. If malloc fails for some reason it rarely will fail because it runs out of memory, it's more likely to fail because you called free on something you shouldn't have called freed on. So you've messed up the whole memory allocator behind the scenes.

That's a good thing to do right there because it very clearly tells you where there's a problem, as opposed to it just seg filtering or bus erroring or it crashing in some anonymous way, and you have not idea how to back trace to figure out where the problem started.

As far as stackDispose is concerned, this is trivial. Although, there's one thing I want to say about it, actually, two things I can think of. I want to go ahead and do the opposite of malloc. This corresponds to operator delete from C++. I want to free s-> elems so that it knows that whatever figure is identified by that address right there should be donated back to the heap. That's just what I mean when I talk about free right here. Some people question whether or not I should go and actually free s itself. They ask whether or not that should happen. The answer is no.

Nowhere during StackNew did you actually allocate space for a stack. You assumed that space for the stack had been set aside already, and that the address of it had been identified to the StackNew function. So you don't even know that it was dynamically allocated. In fact, the sample code I wrote last time declared a stack called s as a local variable. So it isn't dynamically allocated at all. So you definitely in this case do not want to do this.

The other thing I want to mention is that regardless of whether or not the stack, at the moment this thing is called, is of depth zero or of depth 4500, the actual ints that are held inside the dynamically allocated rectangle of bytes, there's no reason to zero them out. And there's certainly no freeing needs held by those integers. I say that because just imagine this not being an int stack but imagine it being a char * stack, where I'm storing dynamically allocated c strings, Freds' and Wilmas' and things like that.

If I did have that, in the case of the char *, I would have to for loop over all the strings that are still held by the stack at the time it's being disposed and make sure I properly dispose of all of those strings. I don't have any of that with the int specific version. But the reason I'm saying this is it's going to have to accommodate that very scenario when we go generic on this thing and just deal with blobs as opposed to integers.

The most interesting of the four functions is the StackPush. And it's interesting not because of the algorithm to put an integer at the end of an array, but the algorithm that's in place to manage the extension of what's been allocated to be that much bigger because you've saturated what's already there.

I chose an initial allocated length of four, so I'm certainly able to push four integers onto the stack and not meet any resistance whatsoever. But if I try to press a fifth in on the stack, it has to react and say, I don't have space for five integers I better go and allocate space for some more, copy over whatever's been pushed, and dispose of the old array to make it look like I had space for 8 or 16 or 1,024 or whatever.

So the implementation, assuming I do have enough space, would be this simple: StackPush. Pushing onto what stack? The one that's addressed by this variable called s. What number am I pushing on? The one that comes in via the value parameter.

Let me leave some space here for what I'll inline as the reallocation part. But if there's enough memory, where down here I can assume that there's definitely enough memory, I can do this:

s-> elems of s-> logLength equals this value

Think about the scenario where the stack is empty. The logLength happens to be zero, which happens to be the index where you should insert the next element.

Once I do this for the next time, I have to go ahead and say, you know what, the logLength just increased by one. Not only does that tell me how deep the stack is, it also tells me the insertion index for the very next push call. It isn't that simple. Eighty percent of the code that gets written here has to deal with the one in two the end time scenarios where you actually are out of space.

If it is the case that s-> logLength == s-> allocLength, then as an implementation you're unhappy because you're dealing with a stack and a value where the value has no home in the stack at the moment.

I'm trying to press on a 7. Let's say that s points to this right here, and the logical length and the allocated length are both 4, and I've pushed these four numbers on there, the client doesn't have to know that there's a temporary emergency. So right here I'm just gonna react by saying, you know what, that 4 wasn't big enough, I want to go ahead and I want to, without writing code yet, I want to basically reallocate this array. Now, I say reallocate because there really is a function related to that word in the standard C library.

In C++ you have to go ahead and allocate an array that's bigger. You don't have to use a doubling strategy; I'm just using that as a heuristic to allocate something that's twice as big. You have to manually copy everything over, and then you have to dispose of this after setting elems equal to the new figure.

It turns out C is more in touch with exposed memory than C++ is. So rather than calling malloc yourself, you can call a function that's like malloc except it takes a value, that's been previously handed back to you by malloc, and says please resize this previously issued dynamically allocated memory block. That is this:

Let me write just one line right here. I want to take allocLength and I want to double it. I could do plus equals 10; I could do plus equals 4. I happen to use a doubling strategy. And then what I want to do is I want to call this function. Let me deallocate these so I have space to write code.

s-> elems equals this function called realloc

There's no equivalent of this in C++. I'll explain it in a few weeks why there isn't. But realloc actually tries to take the pointer that's passed in and it deals with a couple of scenarios. It sees whether or not the dynamically allocated figure can be resized in place, because the memory that comes after it in the heap isn't in use. There's no reason to lift up a block of memory and replicate it somewhere else, to resize it if the part after the currently allocated block can just be extended very easily in constant time. The second argument to realloc is a raw number of bytes. So it would be:

s-> allocLength times size of integer.

I see a lot of people forgetting to pass in or forgetting to scale this byte size event. Even though that they know to do it with malloc, they forget with realloc for some reason. It takes this parameter right here, it assumes it's pointing to a dynamically allocated block, and it just takes care of all the details to make that block this big. If it can resize it in place, all it does is it records that the block has been extended to include more space and it returns the same exact address.

So that deals with the scenario where this is what you have, this is what you want, and it turns out that this is not in use so it can just do it. So it took this address in and it returns the same exact address.

Well, you may question why does it return an address at all? In this case it doesn't need to. However, it may be the case that you pass that in and you want to double the size or make it bigger, and that space is in use. So it actually does a lot of work for you there. It says, okay, well, I can't use this so I have to just go, and it really calls malloc somewhere else on your behalf. Just assume that's twice as big as this. Whatever byte pattern happens to reside here is replicated right there. The remaining part of the figure isn't initialized to anything, because it's uninitialized just like most C variables are. It actually frees this for you and it returns this address.

So under the covers, beneath the hood of this realloc function, it just figures out how to take this array and logically resize it, and preserves all the meaningful content that's in there. If it has to move it it does free this. It turns out realloc actually defaults to a malloc call if you pass in null here. So technically you don't need the malloc call ever. That actually turns out to be convenient when you have entered a processes that have to keep resizing a node, and you don't want a special case it and call it malloc the very first time, you can just call it realloc every single time when the first parameter is null on the very first iteration.

It's very easy to forget to catch the return value. If you forget to do that then you refer to s-> elems captures the original address, which may have changed, and you now after this call may be referring to dead memory that's been donated back to the heap manager. So it's very important to catch it like this. If realloc fails it'll return null. So I'm gonna put an assert right here.

The one thing about realloc that's neat is that if you don't want to just end the program because it failed, it actually, if it returns null it won't free the original block, it would only return null if it actually had to move it. Actually, that's not true. If it can't meet the realloc page in request it'll just leave the old memory alone and return null. In theory, you don't have to assert. And in the program here you could just check for null, and say, okay, well, maybe I won't resize it, maybe I'll just print a nice little error message saying, I actually cannot extend the stack at this time, my apologies, please do something else.

We'll learn a lot about how malloc and realloc and free all work together. They're implemented in the same file. They're implemented in the same file because even though that the actual blob of memory doesn't expose its size to you, like in a dot length field like in Java, somehow free knows how big it is. Well, there's cataloging going on behind the scene so it knows how much memory to donate back to the heap every time you call free. But I don't want to focus on that.

This right here, it doesn't get called very often. This doubling strategy is popular because it only gets invoked as a code block one out of every two to the n calls once you get beyond 4. Because of the doubling strategy [inaudible] only comes up once every power of 2; 512 wasn't big enough, okay, well, maybe 1,204 will be. And you have a lot of time before you need a second reallocation request.

There are a couple of subtleties as to how this is copied. In this example right here, all of these little chicken scratch figures right there, they correspond to byte patterns for integers. So when I replicate them right there, I trust that the interpretation of those integers right there will be the same right there. So realloc is really moving my integers for me.

If these happen to be not four integers but four char *'s, that means the byte patterns that were here would have actually been interpreted as addresses. When it replicates this byte pattern down here, it turns and it replicates the addresses verbatim. This would point to that; that will point to that; this will point to that; this will point to that. Do you understand what I mean when I do that?

When I dispose of this it doesn't go in and free the pointers here. It doesn't even know there are pointers in the first place so how could it free them? So as this goes away, and these all point to where other pointers used to be pointing, you don't lose access to your character strings.

This is certainly the most involved of all four functions.

Student: If you could explain the assertion again.

Instructor (Jerry Cain):All assert is, for the moment just pretend that it's a function. It takes a boolean. Its implementation is to do absolutely nothing and return immediately if it gets true, and if it gets false its implementation is to call exit after it prints an error message saying an assert failed online, such and such, in file stack dot c.

Student: So actually [inaudible].

Instructor (Jerry Cain):S-> elems ? null. You want to assert the truth of some condition that needs to be met in order for you to move forward. If realloc fails it will return null. You want to make sure that did not happen. That's why there is a not equals there as opposed to a double equals.

This isn't technically a function. We'll rely on that knowledge later on. It's technically what's called a macro. It's like a #define that takes arguments. But, nonetheless, just pretend for the moment that it works like a function.

Student: If the realloc has to move the array is it now or anytime?

Instructor (Jerry Cain):Yeah. The question is if realloc has to actually move the array it is time consuming. It's even more time consuming than o of m. It actually involves not only the size of the figure being copied, but the amount of time it takes to search the heap for a figure that big. So it really can, in theory, be o of m, where m is the size of a heap.

Let me just quickly, just for completeness, write stackPop. It turns out it's not very difficult. The most complicated part about it is making sure that the stack isn't empty.

This continues on this board right here. I have this thing that returns an int stackPop. I'm popping off of this stack right here. No other arguments assert that s-> logLength is greater than 0. If you get here then you're happy because it has something you can pop off. So go ahead and do the following:

s-> logLength -- and then return s-> elems of s-> logLength

The delta by one and the array access are in the opposite order here. I think for pretty obvious reasons. You're effectively control z'ing or command z'ing the stack to pop off what was most recently pushed on. So if this came most recently, you want to say I didn't even do that. Oh, and by the way, there's the element that I put there by mistake. That's what I mean.

You could say you see this as demanding a reallocation request and going from 100 percent to a new 100 percent, where the old 100 percent was actually 50 percent. You may ask whether or not if you fall below a 50 percent threshold that you should reallocate and say, oh, I'm being a memory hog for no good reason and donate it back to the heap. You could if you want to.

My understanding of the implementation of realloc, because it wants to execute as quickly as possible, it ignores any request to shrink an array. As long as it meets the size request, it doesn't actually care if it allocates a little bit more. So it's like, oh, the size is 100 bytes and you just want 50. Okay, then only use 50 but I'm gonna keep 100 here because it's faster to do that.

With regard to the stackPop prototype right there, it sounds like a dumb observation, but it'll become clear in a second when we go generic. This particular stackPop elects to return the value being popped off. I can do that very easily here because I know that the return value is a four-byte figure. If this were a double stack I would just make that leftmost int up there double and it would just return an eight-byte figure.

When I go generic, and I stop dealing with int *'s and I start dealing with void *'s, I'm gonna have to make the return type either void *, which means I return a dynamically allocated copy of the element, for reasons that'll become clear in a little bit I'm not gonna like that, or I have to return void and return the value by reference by passing in an int * or a void * right here. That will become more clear once I actually write the code for it. But we take advantage of a lot of the fact that we know that we're returning a four-byte figure here so the return type can be expressed quite explicitly as in int.

So now what I want to do is I want to start over. And I want to implement all of this stuff very generically. And I want to recognize that we're trying to handle ints, and bools, and doubles, and struct fractions, and actually the most complicated part of it are char *'s because they have dynamically allocated memory associated with them.

Let me redraw stack dot h. And we are going completely generic right here. Most of the boilerplate is the same. Typedef struct, and I don't want to commit to a data type, elems. Now, think about what I lost. I lost my ability to do pointer arithmetic without some char * casting. I also lost intimate knowledge about how big the elements themselves are. So I can't assume the size of int anymore because it may not be that; it probably won't be. So I'm gonna require the prototype of StackNew to change, to not only pass in the address of the stack being initialized, but please tell me how big the elements that I'm storing are so that I can store it inside the struct.

The logical length versus the allocated length and the need to store that, that doesn't change. I still want to maintain information about how many of these mystery elements I have. I also want to keep track of how much I am capable of storing, given my current allocation. And there's gonna be one more element that we store on a byte. So I'll leave that piece of spencil for the next 15 minutes and we'll come back to it.

The prototype of the functions change a little bit. StackNew, same first argument, but now I take an elemSize.

Void * stackDispose

stack *s (That doesn't need to change. It's mostly the same, although, we'll have something to say about what happens when it's storing char *'s.)

Void stackPush

stack s

And then I'm gonna pass in void * called elemAddr.

I can't commit to a pointer type that's anymore specific right there 'cause it may not be an int *, it may not be a char **, it may not be a structFraction *. It just needs to be some address that I trust because the information that I am holding is pointing to an s-> elemSize figure. So there's that.

This is the part that freaks people out. This is what I'm going to elect to do, I'll talk about the alternative in a second. But when I pop an element off the stack, I want to identify which stack I'm popping off of, and I also want to supply Address. This is me supplying an address. I'm actually gonna identify a place where one of my client elements, that was previously pushed on, should be laid down. It's like I'm descending a little basket from a helicopter so somebody can lay an integer or a double or something in it so I can reel it back up to me. So void * elemAddr. And that's all I'm gonna concern myself with.

Student: Where is the struct named stack?

Instructor (Jerry Cain):I'm sorry, I just forgot it. It's right there.

So 70 percent of the code I'm gonna write it's all gonna be the same. It's gonna be slightly twisted to deal with generics as opposed to ints. But rather than using assignment, which works perfectly well when you're taking one space for an int and assigning it to another int, we're gonna have to rely on memcopy and things like that.

StackNew is not hard.

Void StackNew

stack *s int elemSize (Same kind of stuff)

s-> logLength = zero

s-> allocLength = four

s-> elemSize = elemSize

s-> elem = malloc

I don't have a hard data type so I can't use size of here, but I have no reason to. I have been told how big the elements are via the second parameter. So four times elemSize, just use the parameter as opposed to the field inside the struct.

I can do a few things to make my life easier. S-> elems better not be equal to null or else I don't want to continue. And the assert will make sure of that. I also could benefit by doing this: assert that s-> elemSize – this one is not as important because it takes a lot of work to pass in a negative value for elemSize. But, nonetheless, it's not a bad thing to put there because if you try to allocate -40 bytes it's not gonna work. That and implementation make sense.

I think even if it makes sense, there's some value in seeing a picture. It takes this stack right there, with its four fields inside at the moment, and let's say I want to go ahead and allocate a stack to store doubles. That means the elemSize field would have an eight right there. I'd set aside space for four of them. The logical length is zero. I'm just making sure this is consistent with the way I've done this. That's right. And then I would set this to Point 2. As far as the stack knows, it has no idea doubles are involved, it just knows that it's a 32-byte wide figure. And it has all of the information it needs to compute the boundary between zero and one, one and two, and two and three.

As far as stackDispose is concerned, stack *s, this is incomplete, but it's complete for the moment given what I've talked about. I just want to go ahead and I want to free s-> elems, and not concern myself yet with the fact that the actual material stored inside the blob might be really complicated. Just for the moment think ints, and doubles, and plain characters. But the fact that I'm leaving space there [inaudible] that this will change soon.

Let me write stackPush right here.

void stackPush

stack *s

void * elemAddr

I'm gonna simplify the implementation here a little bit by writing a helper function. I could have written the helper function on the int specific version but just didn't.

Up front, if it's the case that s-> logLength == s-> allocLength, then I want to cope by calling this function called stackGrow. And you know what stackGrow's intentions are. And I'm just gonna assume that stackGrow takes care of the reallocation part. I'll write an int second, that's not the hard part. Once I get this far, whether or not stackGrow was involved or not, I want to somehow take the s-> elemSize bytes that are sitting right there and write into the next slot in memory that I know is there for me, because if it wasn't this would have been called.

So what has to happen? Let me refer to this picture. Let's just assume that this picture is good enough, because I obviously have enough space to accommodate this new element. Let's say that I have three elements. So that these have been filled up with interesting material. And I somehow, in the context of that code over there, have a pointer to some other eight-byte figure that needs to be replicated in.

This is gonna be the second argument to memcopy. This right there is gonna be the third argument. The only complexity is computing and figuring out what the first argument is. It has to be that. It doesn't need to know that it's a double *, or a struct with two ints inside *, or whatever. It just needs to actually get the raw address and replicate a byte pattern. No matter what the byte pattern is, as long as it's the same in both places, you've replicated the value.

So the hardest part about it is doing this:

void * target = char * s-> elems + s->

logLength times s->elemSize

This is why I demanded all of this stuff to be passed to my constructor function so it was available to me to do the manual pointer arithmetic later on.

I don't think I messed this up. LogLength is right there for the same reasons it was in between the square brackets for the int specific implementation of this.

Then what I want to do is I want to go ahead memcopy into the target space whatever's that elemAddr. How many bytes? This many, s-> elemSize. Then I can't forget this, this was present at the other implementation as well. I have to note the fact that the logical length just increased by one. So that's how I managed that.

Do you guys see just the bytes moving on your behalf in response to this function?

There's a question back there.

Student: The char * after void target equals?

Instructor (Jerry Cain):This right here, remember that s-> elems, unless I made a mistake, is typed to be a void *. So you can't do pointer arithmetic on the void *, so the trick – there's actually two tricks. I opt with this one because I'm just more familiar with it, you can either cast it to be a char * so that pointer arithmetic defaults to regular arithmetic. This as an offset it's still the offset, it just happened to involve multiplication. It is itself implicitly multiplied by size of char, which as far as multiplication is concerned, is a no op.

I have also seen people, I think I mentioned this before, I've seen people cast that to be an unsigned long, so that it really is just plain math, and then they cast it to be void *. Pointers and unsigned longs are supposed to be the same size on 32-byte systems. A long is supposed to be the size of a word on the register set. So I've seen that as well. You can do whatever you want, I just – all of my examples use the char * version so that's why I use that one.

The stackGrow thing. The reason I want to do that is not because of the mem copying or the void *, I just want to explain what a student asked two seconds before class started. You're kind of already getting the idea that the word static has like 85 meanings in C and C++. Well, here's one more.

When you see the word static, decorating the prototype of a C or a C++ function, not a method in a class just a regular function, such as static void stackGrow, and it takes a stack *s. What that means is that it is considered to be a private function that should not be advertised outside this file. So in many ways it means private in the C++ sense. The technical explanation is that static marks this function for what's called internal linkage.

You know how you're generating all these dot o files, when you type make all this stuff appears in your directory, some of them are dot o files. I'll show you a tool later on where you can actually look at what's exported and used internally by those dot o files.

StackPush, and stackNew, and stackDispose are all marked as global functions, and that the symbols, or the names of those functions, should be exported and accessible from other dot o files, or made available to other dot o files. Something like this is marked as what's called local or internal. And even though the function name exists, it can't be called from other files. That may seem like it was a silly waste of time, but it really is not.

Because you can imagine in a code base of say one million files, it's not outlandish believe it or not. Think Microsoft Office, the whole thing, probably has on the order of hundreds of thousands of files, maybe tens of thousands, I don't know what it is, more than a few. You can imagine a lot of people defining their own little swap functions. And if they're not marked as internal functions at the time that everything is linked together to build Word or Excel or whatever, the linker is gonna freak out and say, which version of swap do I call? I can't tell. Because it has 70 million of them. But if all 70 million are marked as private, or static, then there's none of those collisions going on at the time the application's built.

This is responsible for doing that reallocation. Assume it's only being called if it understands that this is being met as a precondition. So it can internally just do this:

s-> elems = realloc

s-> allocLength times s-> elemSize

That's the cleaner way to write it. It makes this focus on the interesting part that's hard to get, and kind of puts this aside as uninteresting.

Algorithmically, the function that's most different from the integer version actually is this StackPop call. This is how I want to implement it:

void stackPop.

I'm popping from this stack right here. I'm placing the element that's leaving the stack and coming back to me at this address. There's no stack shrink function to worry about. So what I want to do is declare this void *, not called target but called source = char * of s-> elems plus s-> logLength minus 1 times s-> elemSize. I forgot to do the minus minus beforehand so I recovered by doing a minus 1 right there.

This is where we're drawing a byte pattern from in the big blob behind the scenes. We're drawing byte patterns from there so we can replicate it into elemAddr. ElemAddr is the first argument. I'm copying from that address right there how many bytes. This many. And then do what I should have done earlier, logLength --. This really should have been the first line, and I shouldn't have the -1 there, but this is still correct.

Do you guys get what's going on here? If you understand the helicopter basket analogy? We're on actually identifying a space where it's safe to write exactly one element so that when the function returns I can go, wow, that's been filled up with an interesting element to me. And I can go ahead and print it out or add it to something or replace the seventh character or whatever I want to do with it.

This used to be int. If I wanted to I could have punted on this right here and just passed in one argument. And I could have returned a void * that pointed to a dynamically allocated element that's elemSize bytes wide. And I just would have copied not into elemAddr, but into the result of the malloc call. With very few exceptions, malloc and strdup and realloc being them, you usually don't like a function to dynamically allocate space for the call and then make it the responsibility of the person who called the function to free it.

There's this asymmetry of responsibility, and you try to get in the habit as much as possible of making any function that allocates memory be the thing that deallocate's it as well. There's just some symmetry there and it's just easier to maintain dynamically allocated memory responsibilities.

It's not wrong it's just more difficult to maintain. It actually clogs up the heap with lots and lots of little void *'s pointing to s-> elemSize bytes, as opposed to just dealing with the one central figure that's held by the stack, and then locally defined variables of type int and double that are passed in as int * and double *'s recognized as void *'s. But because we laid down the right number of bytes, as long as everything is consistent we get back ints and doubles and things like that.

Student: In this case do you have to with our elemAddr [inaudible] right?

Instructor (Jerry Cain):Yeah. I'm not actually changing elemAddr I'm just changing what's elemAddr. So let me actually make a sample call to this.

Suppose I have a stack s. And I called StackNew with & of s, and I pass in size of int. That means that I have this thing, that's my stack and I'm supposed to just take it as a black box and not really deal with it. But I know behind the scenes that it has a 4 there, and maybe my stack has 14 elements in it, and it can accommodate 16 elements. And that it points to this thing that has not 2 or 4 or 8, but 16 elements in it. And I'm like, you know what, I did all this work, and I pushed on 14 elements right there, but I'd like to now pop off the top element.

When I do this stackPop, I have to pass in that. I have to pass in the address of an integer. So the stackPop call has a variable called elemAddr that points to my variable called top, and it relies on this variable to figure out where to write the element at position 13. It happens to reside right there. The memcopy says where do I write it? I write it at that address right there. You would replicate this byte pattern in the top space, this returns, and I print out top and that corresponds to the number 7,300 or something like that.

If I really wanted to change this void *, I don't change the void * anywhere, I just use it as sort of a social security number or an ID number on the integer. If I really wanted to change this you'd have to pass in a void * *. There are scenarios where that actually turns out to be what you need, but this is not one of them.

Student: For this pop, let's say that this function doesn't work, that it's popping the wrong stuff. How do we test to find out if we pop [inaudible]. Like, for instance, let's say you wanted to pop something that was an integer, and you popped [inaudible], and then what if there was something wrong with the resizing, can we set that in that we just pop to like null or something and then –

Instructor (Jerry Cain):You could. If you really want to exercise the implementation of the stack to make sure it's doing everything correctly, you'll write these things, you may have heard this, what are called unit tests. Which are usually implemented – there's two different types of unit tests, there's those that are written on the dot c end, which know about how everything works, and then those that are written as a client to make sure that it's behaving like you think it should. If that were the case, and you wanted to protect against that, you could write all these very simple tests to make sure that things are working.

Because in unit tests, as a client you might for loop over the integers one through a million. You might actually set the initial allocation length not to be four, but to be one, so that you get as many reallocations as possible. You could for loop and push the number one through a million on top. In a different function, you could actually pop everything off until it's empty and make sure that they are descending from one million down to one. And if it fails, then you know that there's something wrong. If it succeeds, you never know that something is completely working, but you have pretty good evidence that it's probably pretty close if it isn't.

You can change the implementation to kind of make sure that all of the parts that are really risky, like the reallocation, and the

--, and the memcopy calls are working, one thing you could do, and Assignment 3 takes this approach a little bit, right here I use this doubling strategy. If I really wanted to test the reallocation business, I could actually do a ++ instead of a times two on the allocation size and to reallocate every single time. Not because you want it to work that way, but because you could just make sure that algorithmically everything else works regardless of how you resize them.

Now, another answer to your question, if I didn't take that approach with the unit test answer, is that when you're dealing with generics in C, and void *'s, and memcopy, and manual guests, you have to be that much more of an expert in a language to make sure that you don't make mistakes.

It's very easy. Think about it this way, I know you're not gonna believe this yet because you haven't coded in C that much, but Assignment 3 you will. I know how most of you program. You write the entire program and then you compile it. And you have 5,500 errors. And you get rid of them one by one, and it takes you like three days, and then it finally compiles. And you run it, and even if it doesn't quite run the way you want it to it rarely crashes. Most people don't even know what the word crash means until they get to 107. You will next week, trust me.

As far as C++ is concerned, because it's so much more strongly typed than C is, it is possible for you to write an entire program, to have it compile, because compilation does so much type checking for you in a C++ program that uses templates, it's quite possible that once it compiles that it actually works as expected. It doesn't happen very often but it's certainly possible.

In a C program, where you're using void *'s, and char *'s, and memcopy, and memmove, and bsearch, and all of these other functions you're gonna need to use for Assignment 3, it's actually very easy to get it to compile. Because it's like, oh, void *, I can take that, yep, that's fine. It just does it on all the variables, and so it compiles. And you're like, good, it wasn't three days, it was one day. And then you run it and it crashes because you did not deal with the raw exposed pointers properly. So that's what makes certainly the first assignment in pure C with dealing with void *'s difficult.

But there's an argument that can be made to say that it's very hard to get all this stuff right every single time you program. I've already told you that right here I forget this s-> elemSize probably one out of every two times I teach this example. That's because it's very unnatural compared to the C++ way of allocating arrays to think in terms of raw bytes. And you think in terms of sizes of the figures, you see the pictures in your head as to how big things should be, so you just remember that number right there but you forget about this. If you forget this right here, it compiles, it runs. If you're dealing with integers then your array is one-fourth as big as it needs to be to store all the integers you want there.

You will learn this and feel it at 11:59 a week from Thursday. All these types of errors, because it's gonna compile and it's gonna run very often. But occasionally it's gonna crash and you're not gonna know why, and you're gonna say, oh, it's the compiler. It's not, it's your code.

Okay. So we will talk more on Wednesday.

[End of Audio]

Duration: 51 minutes

ProgrammingParadigms-Lecture07

Instructor (Jerry Cain):Hey, everyone. Welcome. I have one handout for you today. It is Assignment 3. It is due next Thursday evening. You'll get Assignment 4 next Wednesday. It'll be due the following Thursday evening. You'll get a written problem set a few Wednesdays from now. It won't need to be turned in. I'll talk about that more when I actually give it out, but it'll provide a collection of written problems that you'll be responsible for for the midterm come the following Wednesday night.

When I left you on Monday, I had really just gotten through what, at the moment, was the full implementation of a generic stack. I've actually made parts of it easier than it really needs to be because we focused on storing ints, and doubles, and characters, and Booleans.

What I wanna do now is put off the implementation for about ten or 15 minutes, look at that again, and think about how we would use it to store a stack – I'm sorry, use a stack – a generic stack right there – to store a collection of strings and print them out in reverse order.

Now, the nonsense code I'm gonna put on the board is effectively gonna just print out the reverse of an array, but it's really in place to illustrate the mechanics of using those four functions when you're storing C strings. That's gonna become very important come Assignment 4 to manipulate C strings, so that's why I want to do this.

So just imagine this right here being your main function. I don't care about Argc and Argv, but I do care about declaring one of these things – and I'll emphasize the fact that it is a string stack – and what I wanna do is I wanna press on four deep copies of these strings right here, const, char*. I'll just say letters is equal to – actually, you know what? Let me not call them letters. We'll just say friends – and I'll set it equal to this array. Now, Bob, Carl, and that'll be enough.

So what I wanna do is I wanna declare one of these stacks. This picture I get right here, according to that typedef over there, isn't very sophisticated. It's 16 bytes of nothing, or nothing meaningful, okay. But I rely on stack new ampersand of the string stack where I pass in size of char*, and all of a sudden now it's getting a little complicated.

I'm gonna ask the stack to basically keep track of the addresses of dynamically allocated C strings. So what I wanna do is I just got this picture. This points to a 16-byte blob. Four bytes – there's no – the depth – the stack is zero, but I have space for four elements. This four right here is really size of char*.

What I wanna do is I wanna for loop from i's equal to zero, i less than three because I wanna go ahead, and I wanna make a copy of each one of those strings. I do. That's char* – I'll call it Copy – is equal to strdup – oops – of friends of i, okay. This is an important enough variable that I actually wanna draw it, copy. On the very first iteration when i is equal to zero, it is set to point two, a deep copy of Al backslash zero in the heap.

What I wanna do – and I think this is the best way to phrase it – is that when you push an element onto the stack, you transfer ownership from you to the stack. The way you do that, based on this right here, is for me to do stack, push, which stack? The string stack that I've just initialized.

There's some drama and controversy over what the next argument should be. Since I am storing char*, that means that these things – even though the stack doesn't know it – that they have to hold as material these four byte character pointers. That means that I have to pass in the address of a char* so it knows to go to that address and copy the four bytes into the stack. Does that make sense to people?

Okay. Because I'm putting the ampersand here, it knows to go and replicate that material right there, so that it points there, increments this to a one. On the very next iteration, I reuse i, and actually destroy and re-declare Copy to no longer point to Al, but as it's re-declared and reinitialized it's set to point up to Bob.

This is Copy on the second iteration. I pass that in. It replicates the material that's in there because it has the address of that size of char* figure so that it could replicate that address right here, and so it's almost like this as a for loop blows up three balloons, okay, with Al, Bob, and Carl's name on it, and then knowingly memcpys the end of the string, or the tail of the string, and actually copies it to the stack behind the scenes. Do you understand what I mean when I say that? Okay. So, I'm really transferring ownership of these three dynamically allocated copies over to the stack.

What I wanna do now is I wanna go ahead and I wanna print these things out. I really wanna ask for ownership back, so I'm going to do this char*, name, and then a four int i is equal to zero, i less than three, i++.

What I wanna do is I wanna ask for the stack to pop off – string stack – and I want it to place the most recently pressed, or pushed, kite string back into the space that's right here, okay. So I want it to – this would have been set up to point to Carl. I want it to replicate this space in my local variable called Name so this ends up pointing to Carl, and the logical length of the entire stack is detrimental to two. Okay, does that make sense?

The way I do that is like this. Then I can do this. This is the equivalent in C of C out. [Inaudible] where this is the string percent. This is a placeholder for the string to be printed, and then I can go ahead and free main. That means that it basically passes the end of the kite string – or the end of the balloon string – to the deallocator, so it goes to the Carl address, or the Al address, or the Bob address, and actually donates that number back to the heap. I wanna be clean about it. I do stack dispose at the end, and I just pass an ampersand of string stack, and that is that, okay.

Now, these ampersands right here typically don't surprise people because I'm dealing with a direct allocation on the stack frame of this function of a thing called string stack, and I have to pass the location of it around, okay.

These ampersands right there very often surprise people. If you do not put them there, for many of the same reasons we saw in previous examples, it would still compile and run, but if you provide this address to stack push, then you're going to get it right. But if you actually don't include the ampersand right there, and you pass in that value right there, it's going to go ahead and replicate BOBY – I'm sorry – BOB backslash zero – as if it's an address, and copy that into the stack frame, okay. That's not what you want. Okay, does that make sense? Okay, very good.

Now, the problem comes – suppose I go ahead and I comment all of this out, or I set this equal to i less than two, or something. Suppose, at the time, I actually call stack dispose. The stack actually has some material that it still owns.

I've been very symmetric in the way that I allocate build up, bring down, and then dispose of, but the stack shouldn't be obligated to be empty, or the client shouldn't be forced to pop everything off the stack before they call dispose. Stack dispose should be able to say, "Okay, I seem to have retained ownership of some elements that were pressed onto me. I would like to be able to dispose of these things on behalf of the client before I go and clean up, and donate this blob of memory back to the heap." Okay.

Many times there's nothing to be done at all. When this thing stores ints, or longs, or doubles, or characters, you don't have to go in and zero them out. That's really not that useful. You do have to be careful about donating back any dynamically allocated resources, or maybe any open files. That's less common, but dynamic memory allocation is certainly more common.

If these things really are owned by the stack at the time it's disposed of, then the stack dispose function has to figure out how to actually pass these things right there to free just like we do right there. Does that make sense to people? Okay.

It's actually very difficult to do that because the implementation of stack dispose doesn't actually know that these things are pointers. It just knows, at best, that they're four-byte figures. It is capable of computing these addresses, and so if the depth of the stack is three, so that those three arrows are arrows that point to elements it's holding for the client. It could pass those three arrows to some disposal function, okay.

Now, this isn't always going to be a simple pointer. This might be a struct with three pointers inside, okay. It might be itself a pointer to a pointer to a struct that has three pointers inside. So, you wanna have some very general framework for being able to free whatever's at those three arrows if, in fact, there's anything freeing needs.

So, what I wanna do here is I want to upgrade this right here to not take two arguments, but to take three arguments, and this is what it's going to look like. This is the upgraded stack new function. Stack new is going to do that. It's gonna take elemsize, and it's also gonna take this, void. Free function takes a void* and doesn't return anything.

The idea here is that I want to pass to the constructor function information about how to destroy any elements that it holds for me when I call stack dispose, okay. Those three arrows – if you're dealing with a stack of depth three, and it's storing strings – it's prepared to pass those three things in sequence. At least, that's what I wanna write code for – those three things in sequence to this function that you write and pass the name of to your stack new function, okay. And it will invoke this function for every single element it holds for you.

Since we write this, we can accept the void*s right there. We interpret them, in this case, to be the char**s we know them to be, dereference them, and pass them to free, okay. Does that make sense to people? Yes? No? Okay.

So, I have to rewrite a few things. I have to actually store the free function as a field inside the struct. That means that this is a picture. We'll actually have this fifth field that points to a block of code that knows how to free things for me, okay.

We have to also handle the scenario where we're storing ints or doubles in the stack, and there actually is no freeing to be done on behalf of those things. So, the client, when they call this new function, they're supposed to pass on the first two arguments as they always have.

If you're storing these base types that have no freeing needs, I expect the client to pass an annul here, and that will be checked for and stack dispose. If you're storing things like char*s, or pointers to structs, or even direct structs that have pointers inside that are pointed to dynamically allocated memory, then you have a meaningful free function placed right here, okay. Does that make sense? Okay.

Let me rewrite stack new. I'm sorry, not stack new. This is easy. Let me rewrite stack dispose. Stack astro s understand that now I'm getting the pointer of one of these five field structures where the fifth field is actually either some null pointer, or a pointer to a legitimate freeing function.

Before I go ahead and dispose of the elems blob, I better check to see whether any – to see whether or not anything complicated is residing within the logLength elements that are still inside. If it is the case that s arrow free function – that's the name I gave to that field up there – I don't want to be too clever the way I do that, so let me say if it's not equal to null, then that means I have to apply this free function as a block of code against those three arrows, or all of the arrows that come up, the manual addresses – the manual the computer addresses of all the things that reside behind the scenes.

You could do this. Four int i is equal to zero, i less than s, logLength i++, I would just do s arrow, free function, char*, SRO elems plus i times s arrow elemsize. Okay. I think people don't usually argue with that because it's something they've seen before, so they kinda trust it. It's not difficult to get that part right when you know you're storing a free function. The part that's difficult to get right is right in the free function itself.

Let's revisit this example now that we know that when we store strings, we actually have to potentially set up the stack, or set up the stack to potentially delete elements for us. That means when we declare a stack called string stack, and we call stack new with size of char*, and I wanna pass in some function called string free – I'm just contriving the name. I do know that it has to match that right there as a prototype. It has to take a void* and return nothing.

I have the responsibility if actually writing the string free function. Well, I have to set up string free to actually accept the void* knowing that it's really a char**. Does that make sense to people? Okay.

Let's revisit this picture. These are the types of things that are gonna be passed to my free function. I need to reinterpret that as something that's at least dereferenceable, okay, and then hop into the actual rectangle, or the box, and take that number and pass it to the free function, okay.

So, as an aside, prior to doing this you would implement string three to take the void* – oops, let's give it a name – Elem, or VP, or whatever you wanna do – and because I'm writing this specifically for the char* stack case, I would just do this. This is really an asterisk. It just came out badly. Okay, that's a good one.

Okay, now what happens if I forget this, and I forget that? Think about what actually gets passed to free. If I don't cast this to at least be a double pointer – and I'm actually casting it to be a char double pointer because I know it's really two hops away as an arrow to characters – and dereference – this dereference is really what matters. It's the thing that takes me from this little fence post right there to the box that's addressed by the fence post, okay.

If I leave that that way, it will pass this address to free. It will pass that address to the free function. It will pass that address to the free function, and that's bad because the first one actually can be passed to free. You shouldn't be doing that, though, because you didn't allocate that block. These two addresses should certainly not be passed to free because they weren't directly handed back to anyone via up call to malloc or realloc, okay.

You don't own these copies of the pointers, but you know that they're char*s, and you're just telling the stack to actually dispose of those things for you because even though the stack isn't empty, you don't need the stack anymore. That's why it's imperative that these things really be there, okay.

If you're storing a stack of ints, or a stack of longs, or a stack of even struct fractions where there's no pointers inside, you would just pass in null there instead, and that would be special-cased away right there, okay. Does that make sense to people? Okay.

If I'm storing ints, or longs, or even struct fractions – which, when we double struct fractions just had two direct ints inside, if there's no dynamic memory allocation involved in the things I'm storing – even if they're structs – I would pass in null. The constant right here is a sentinel meaning there's no free function needs, okay, and it's special-cased, and would be observed to be null right here, and circumvent this for loop, okay.

Just to make sure, let me ask some questions. Some of them were easy, but I wanna make sure you believe the answers. You understand why I for loop up to logLength and not allocLength, right? I have no business asking the stack to free things beyond the boundary between what's in use and what's not in use, okay. I have no reason to trust that anything meaningful is in that extra space. In fact, I know it isn't.

How come I don't free the element using this function right here just before stack pop exits? Do you understand what I mean? Like, if the stack is saying, "Oh, they want an element back. I better return this." Why isn't the stack applying the free function to the top element before it returns it? The way I framed it there it kinda sounds like an idiotic question. But you're not so much – you're not really transferring a copy of the string back. You're transferring ownership of the original string back to the client. You're taking this pointer and using memcpy to replicate it in client-supplied space.

So, if you do that and they have an alias to a pointer that you have an alias for, and then you apply the free function to it, you're killing the string one instruction after you actually give it back to the client, okay. Does that make sense? Okay, that's great.

Even if all the code makes sense, just be sensitive to the fact that the compiler does not help you out as much as we'd like. So, you really have to be very thoughtful about the placement of the ampersands, and the double asterisk versus the single asterisk, and whether or not you have to dereference a char** cast as opposed to not dereferencing a plain char* cast, okay.

The number of hops really matters, okay. And you always want to interpret the void*s to be the types of addresses you really know them to be, okay, because if you don't the composite is gonna let you do whatever you wanna do. And – well – I mean, in theory a compile time you can get away with a lot, but at run time you never get away with anything, okay. Does that make sense to people? Yeah?

Student: So, the free function will understand it's not just one character we're looking at; it's the whole string and the element?

Instructor (Jerry Cain):Well, that's not so much – that has nothing to do with string so much as it has to do with malloc and free, but the addresses that reside in this space right here, they were created using this function called strdup. I think the code's still up on the board right there, and strdup actually relies on malloc to allocate the memory.

Behind the scenes, even though it's unexposed to us, it actually keeps track of exactly how much memory is part of that blob. So, as long as you pass the leading address to it, it goes to a file where everything is kept track of, and it looks – it compares that address to something in a symbol table, or something else to recover the actual allocation size so it knows exactly how much to donate back to the heap. Does that make sense? Okay.

Any other – the question way in the back.

Student: In the first line of int main, should that be a double char* [inaudible]?

Instructor (Jerry Cain):[Inaudible] it absolutely should be. This should be a double star. I meant to do this. Sorry. That's the way it is always in my sample code. I just forgot to do it here. Sorry about that. Was there another question that flew up over here? Yeah.

Student:[Inaudible]

Instructor (Jerry Cain):But I did, actually. This – there's a malloc that happens as part of that strdup. So, every single string has to be either freed by the stack itself as part of stack dispose, or if it comes back to me because I asked for via stack pop, and after I'm done with it I have to probably dispose of it. So there has to be a one-to-one correspondence between every call to strdup and every call to free.

In an ideal world where you only dispose of empty stacks, all the free calls would come in the client side. But if you ever dispose of a stack that still holds on to its elements – or is holding on to a subset of the elements – then the number of free calls is going to be distributed between the stack and the client. Does that make sense, okay. That's good. Okay.

So, what Assignment 3 is all about – it actually turns out that Assignment 3 used to be a very, very difficult assignment, but I started doing this in lecture about, like, I don't know, like three and a half years ago. And then all of a sudden, things became much more clear when it come assignment time because the implementation you write for the first half of Assignment 3 is really just an extension of this.

Rather than actually just dealing with push and pop as operations – those are the dynamic operations we're memcpy'ing those on – I want you to generalize it so that you can insert anywhere, and you're familiar with that from a data structure standpoint using the capital V vector from 106 or the lowercase v vector from the STL that you used in Assignment 1 and 2, okay. Does that make sense?

There are two things that I just wanna mention before I formally put this material to bed, and I wanna talk a little bit about the implementation of malloc, and free, and realloc and how they work. It's actually very interesting, I think. But I have to talk about two other functions that come up during the implementation of Assignment 3. I should just talk about them.

Let's – this is okay – I wanna write a function that's very similar to swap. I wanna write a function called rotate. This is actually imitates a function that's provided as part of the STL library, but I'm gonna write it in pure C, and I'm gonna pass in three void*s. Void*

– I'll call it front – void* – I'll call it middle – and void* – I'll call it end. And what I wanna do – I'll use this board to draw a picture – is I want front, middle, and end to actually be sorted pointers that point to various boundaries inside an entire array.

So let's assume I have an array of 50 integers, okay, and for whatever reason, I want to move the front four all the way to the back, okay. Does that make sense to people? So, think of it as, like, a bookshelf with 50 books on it, and for whatever reason the front four are out of alphabetical order, and you've decided to take them out as a chunk, and move them to the back, but in the process you have to slide 46 books forward. Now drop the word "book" and use the word "int" and that's what I wanna do.

The intent of this rotate function is if it's given the absolute opening address of the entire figure, it's given the midpoint that separates front from back – even though they're not equal sized – and then end is actually passed the end iterator in the C++ stance, but it's really the address of the first byte that has nothing to do with the array.

I can manually compute the number of bytes that's right there. I can manually compute the number of bytes that's right there from these three void*s. This is an implementation, needs to know nothing at all about the fact that that happens to be 50 ints over in that drawing. It could've been 200 characters, or it could've been 100 shorts, and it still should do the byte rotation in exactly the same way, okay.

The slight complication here that did not come up in the swap implementation is that this right here has to be written to temporary space. You're familiar with that idea from the swap implementation. Then this right here has to be memcpy'd – although that won't be the function we'll use. We'll see in a second why. This has to be memcpy'd from right here to right here. Does that make sense? Okay.

The problem with that is that – unlike all the other examples we've dealt with – the source region – this space, and this space right there – actually overlap, or can potentially overlap. Does that make sense to people?

The implementation of memcpy is brute force. It carries things four bytes at a time, and then at the end does whatever mod tricks it needs to copy off an odd number of bytes, but it assumes they're not overlapping, okay. When they're overlapping, that brute force approach might not work.

Suppose I wanna copy these first five characters – and this is meaningless, and this is meaningless. What memcpy would do – if I wanted to copy these five characters to right there – memcpy is actually quite careless – and it doesn't do exactly t his, but it does more or less this – where it would actually copy the A right there, and then copy the B right there, and then copy the C right there, and then copy the D right there, and copy the E right there, except that you've trounced on the C, D, and E before you had a chance to copy it. Does that make sense to people?

Now, memcpy could figure it out. It could actually check the start address – I'm sorry – the target address and the source address – and it could copy either from the front to the back or the back to the front, whichever direction is needed to ensure that it doesn't actually trounce over data before it's copied. Memcpy said, "I don't wanna bother with that. I wanna run as quickly as possible, and I want the client to take responsibility of only calling memcpy when, in fact, he or she knows that there's no overlap."

If they don't know if there's gonna be overlap they have to use a version of the function that's slightly less efficient, but does the error checking for them. Does that make sense? It has the same exact prototype – that shouldn't surprise you – but it's not called memcpy. It's called memmove.

I don't know why they use "move" as opposed to "copy." Probably because it's supposed to imply that it's actually shifting somewhere in – if you're dealing with overlapping ranges that you're really just moving bytes in a direction just by a little amount as opposed to really relocating them, okay.

So, the implementation of this has to be sensitive to that. What I wanna do is I wanna compute a few values. I wanna do int, front, size is equal to char*, middle minus char* front. Now, you look at that, and that looks a little weird. I am subtracting one void* from another. For the same reasons C does not allow pointer arithmetic – I'm sorry – pointer addition, it doesn't allow pointer subtraction either. Pointer subtraction you didn't deal with too much in 106B or 106X, but it is a defined, legal operation.

What it's supposed to do when you subtract one pointer from another – you may think that it returns the number of bytes that sits in between them. That's not true. If they're strongly typed to be int*s for instance – if you do pointer subtraction between two int*s it's supposed to return the number of ints that fit in between the two addresses, and that's consistent with the way that pointer addition works, okay. Does that make sense to people?

So, what I'm doing here is I'm – it's the same hack. I'm casting both of these things to be char*s so that pointer subtraction becomes regular subtraction, and I'm given the physical number of bytes that reside between that and that right there. Does that make sense? Okay.

I wanna do the same thing with end and middle so at least I know how big the two regions are. What I can do now is I can declare a raw buffer, just like I did with the generic swap function, char buffer, and I can allocate it to be of size front size. Again, this isn't ANSI standard, but it works on our compiler, and it's so much nicer that I'm just gonna do it.

So now I have a character buffer that's just as big as this thing is here, in terms of bytes. So maybe that's set aside right here because I said those were ints – if I draw it even close to scale – it will be of size 16 right there. And then I use memcpy.

I actually prefer to use memcpy, and I want you to use memcpy if you know you have the option to. I wanna memcpy into this buffer from front, front size so that if this as a figure happens to reside there, it's replicated right there. And if these are four ints, then this will eventually be able to stand in as four ints when it's placed in integer space.

Then I wanna take this and I wanna slide it down. When you call memcpy your heart's in the right place, but you're dealing with two overlapping figures so you wanna call – not memcpy – but memmove with the same signature, okay. That would be this – memmove. I wanna copy to front from middle, and I wanna copy back size, okay. Does that make sense to people?

Now, if mid is very close to the end, and it's beyond the 50 percent point, then it turns out that memmove didn't buy us very much because there's no overlapping figure – I'm sorry – there's no overlapping between the source region and the destination region.

But you can't – in a general sense – actually anticipate that, okay. You could actually look at how much closer – whether or not middle is closer to front or end – and then call one of two versions that only called memcpy, but then all you're doing is the error checking that memmove would do for you. So I'd rather just deal with one implementation, okay. Make sense? Okay.

The only complexity of the last line is getting this right here into the last front size bytes. I don't actually have a target pointer for this, but what I'll do is I'll memcpy – that's okay because I'm copying from the buffer – I'll do char*, end minus front size. If from here to here is front size, then from there to there is front size. I wanna copy from the buffer, and the size of the buffer is front size, okay. Does that make sense?

So, you only call memmove if you have to, okay, because you know that there's a very reasonable chance that the two – the source region and the destination region – will be overlapping. But I want you to call memcpy if you know you're able to because it's more efficient, and when you're starting to write systems code like this you actually do have to think a little bit about – more about – efficiency than you did in 106b.

People argue, "Well, why don't I just call memmove all time because then I can just forget." You're right. You could, but I'd rather you not. So I actually wanna pretend that memmove actually blows the computer up if the two regions don't overlap, okay, because I want you to call memcpy if you know you can, okay. Does that make sense to people? Okay.

The only other function I have to mention briefly – and I actually have plenty of time to do it so I'll just – but I just wanna go over the prototype of the generic quick sort function that exists in the C language, okay.

When you're sorting, there's not key. There's just the array, the element size, the number of elements, and then the comparison function is still relevant. You know how B sort only takes five arguments? Well, Q sort only takes four. It punts in the key. Everything internally is a key compared to everything else, and it just uses the comparison function to guide it in doing all these generic swaps behind the scenes.

I do want you to use Q sort. I just wanna put the implementation up on the board so that I can say I've formally covered it, but you're all familiar with just sorting in general. Quick sort happens to be a very fast sorting algorithm. It has this as a prototype. Q sort takes a void* called base. It takes an int called n – or size, actually – an int called elemsize, and then it takes a comparison function that knows how to compare two generic addresses. And there's that. That's the prototype for it.

If you want more detail – this is even relevant in some degree to Assignment 2 – if you happen to be stuck on using B sorts and you just wanna use some more details, you can – at the command prompt – type in MAN Q sort, and you will get more information about Q sort than you care to get. But nonetheless – at the top – will remind you what the prototype is, and what all the arguments are named so you know which ints correspond to which.

You can do MAN on B search – MAN is short for manual so it just basically gives you a textual documentation of that function. Also, memcpy, memmove – and I think there's some other ones – malloc, realloc, free. These are the types of functions that are exercised aggressively by Assignment 3 so you'll definitely want some resource to go to if it's 5 a.m., and you're working on it, and there's nobody around, okay. Does that sit well with everybody? Okay.

So you have plenty of time to do Assignment 3. It has turned out to be – it is – I won't say it's easy by any stretch of the imagination because I don't have – there's no advantage to me saying that, but it is actually a little bit less work than Assignment 2. And people always start on it with a lot of enthusiasm because they think there's a lot of work to be done – and there actually is – but there's a clear list – a to do list of things – there's like 13 functions that have to be implemented, and I provide all kinds of unit tests to exercise all of these things, okay.

And you will just make very piecemeal progress on a nightly basis, okay, so that you can get it done in one or two nights if you actually know what's going on, okay. So just give yourself a little bit of a buffer in case you have some gotchas that come up while you're implementing. But it is – usually surprises people that it's not as much coding as Assignment 2 is, okay.

What I wanna do now is I have ten minutes. I just wanna give you a little preamble to the more – the more involved lecture I'm gonna give on Friday about the implementation of malloc, and realloc, and free. Let me draw what – for the first time of many –is gonna be my generic drawing of all the memory.

Here's RAM, and since we're dealing with a – since we're dealing with an architecture where longs and pointers are four bytes, that means that pointers can distinguish between two to the thirty-second different addresses. That means that the lowest address in memory is zero – which is that null that you're starting to fear a little bit – and then the highest address is two to the thirty-second minus one, okay.

Whenever you call functions, and the function call forces the allocation of lots of local variables, those local variable – or I'm sorry – the memory for those local variables is drawn from a subset of all RAM called the stack. I'm gonna draw that up here. I drew it a little bit bigger than I needed to, but here it is. The stack segment is what this thing is called.

It doesn't necessarily use all of the stack, but for reasons that will become clear – and there's even a little bit of intuition, I think, as to why it might be called a stack – when you call main you get all of its local variables, and they're alive and they're active, okay.

When main calls a helper function it's not like main's functions – main's variables – go away. They're just temporarily disabled, and you don't have access to them – at least not via the normal variable names, right. So main calls helper, which calls helper helper, which calls helper helper helper, and you have all of these variables that are active – I'm sorry – that are allocated, but only the ones on the bottom most function are actually alive and accessible via their variable names.

When helper helper helper returns, you return back to the local where helper helper has local variables, and you can access those, okay. So basically, when a function calls another function, the first function's variables are suspended until whatever happens in response to the function call actually ends, okay. And it may itself call several helper functions, and just go through lots of – a big code tree of function calls before it actually returns a value, or not, okay.

What happens is that, initially, that much space is set aside from the stack segment to just hold the main's variables – whatever main's local variables are. And when main calls something, this threshold is lowered to there to just make sure that not only is there space for main's variables set aside, but also for the helper function's variables, okay.

And it goes down and up, down and up, every time it goes down it's because some function was called, and every time it goes up it's because some function returned, okay. And the same argument can be made for methods in C++.

It's called a stack because the most recently called function is the one that is invited to return before any other ones unless it calls some other function, okay. That's why it's called a stack.

We'll get to this – we'll probably spend two or three lectures talking about not only how this thing's formatted, and how variables are laid out, but also how assembly code actually manipulates the information in here. And assembly code even actually decrements and increments this boundary for us in response to function call and return.

What I wanna focus on is this other block of memory called the heap segment. The fact that the heap and the stack are date structures from cs106b is actually irrelevant here – mostly irrelevant, okay.

Heap in this world doesn't mean like a priority cube back in data structure. It really means blob of arbitrary bytes that this is the lowest address, this is the highest address of the heap segment as opposed to this segment right here, which is completely managed by the hardware – by the assembly code, which actually happens to be down here, okay – this right there, this boundary, and that address, and that address is admitted to software – software that implements what we call the heap manager, okay.

And the heap manager is software – it's code. The implementation for malloc, realloc, and free are all written in the same file, and they basically manage this memory right here, okay.

So, every time you call malloc of 40, it goes, and virtually finds a block of size 40 in this, and seemingly returns the lead address of it, okay. If you haven't freed 40 yet, and you go and allocate space – or malloc a request for 80, it might draw it from here – it's a little bit more organized than this. It doesn't just draw it from a random location, but malloc would return that.

If you realloc this pointer right here, and you ask for it not to be 40 anymore, but to be 100 because of the way it's drawn it actually might just extend this to be a blank 100, and not touch the bytes that are up top – up front. If you reallocate this, and you ask for 8,000, and it doesn't happen to have 8,000 minus 80 bytes between that boundary and that boundary, it might go and allocate a blob that's 8,000 bytes, copy this over, and copy it right – and then free this right here.

So this really is a sandbox of bytes, and whether they're integer arrays, or char* arrays, or struct fraction arrays, it's all immaterial to malloc. It only allocates things in terms of numBytes requests, okay. Does that sit well with everybody? Okay.

So what I wanna do is I wanna be a little bit more organized in the way I demonstrate how malloc, and realloc, and free work because it isn't that rain pell mell about allocating bytes. It does have some normative process that's followed behind the scenes so it can be as efficient as possible on your behalf. Malloc, and free, and realloc are actually called enough – either directly or via things like operator new, and operator delete, or strdup, or your constructors, or whatever, but it wants to make them run as quickly as possible.

So what I'm gonna do is I'm gonna explode this picture to this board over here – actually, this one's better – but I'm gonna emphasize the fact that it is one big linear array of bytes. And so, rather than drawing it as a tall rectangle, I'm gonna draw it as a very wide rectangle, and make it clear that this address right there is that one right there, and this address right there is that right there, okay. Does that make sense to people?

So I go ahead and I declare void* A is equal to malloc of 40. This isn't exactly what happens, but it's pretty close. It will usually search from the beginning of the heap, and look for the smallest – I'm sorry – the first free block of memory that's able to accommodate this size request.

And initially, since the entire heap is available – what'll happen is it'll do whatever accounting behind the scenes as is necessary to clip off the first 40 bytes, record that it's in use somehow – we'll talk about that in more detail on Friday – and return the address of that right there. Does that make sense? Okay.

Next line is this. Malloc of 60 able – take this least – this naïve approach, and that's what it does very often. It just scans from the beginning of the heap, and find the first open block that's able to meet this request. It sees that that's in use. It's able to quickly hop here – we'll see why on Friday – and say, "Okay, this entire thing is free. That's certainly bigger than 60. So I will do that, and then return that address." Okay.

Punting on realloc for a second, if I go ahead and call free on A it will go in because this address – this arrow right there – the tail of it is held in the A variable.

It will go and remove the halo around this memory – it doesn't clear out the bit patterns because the bit patterns are supposed to stop mattering – and then donates it back. So if I do this, void* C is equal to malloc, and I'll [inaudible] notches about it, and I say 44, it will look at this block right here. It will record it, and note that it's a 40 byte free block that could've been used had this number been less than or equal to 40, but since it isn't it has to hop over and consider this right there. So it will clip this off and return that address and bind it to C.

If on the very next line I do this – some implementations actually carry off where the last search ended, but the way I'm talking about it, it always searches from the beginning, okay. This time this block is big enough to meet that size request so it might clip this off, record that it's 20 bytes wide, and return that pointer again, okay. Does that make sense?

Entirely software managed with very little exception, okay – and I say exception because the operating system and what's called the loader has to admit to the implementation what the boundaries of the stacks of the heap segment are – but everything else is really frame in terms of this raw memory allocator, okay.

As far as realloc is concerned, if I pass this address to realloc, and I ask it to become bigger, it'll have to do a reallocation request, and put it somewhere else – probably right there, the way we've been talking about it. If I didn't ask for that to be realloced, but I asked for this to be realloced – to go to 88 – it would just extend it in place.

What's gonna happen – and we'll be more detailed about this come Friday – is that there really is a little bit of a data structure that overlays the entire heap segment, okay. It is manually managed using lots of void* business.

In a nutshell – let me actually, in the final ten seconds here – let's say that this is the heap again – and just to emphasize what's been allocated and what hasn't been, let's say that this has been set aside. Let's say this has been set aside, and let's say that this has been set aside. The data structure that's more or less used by the heap manager overlays a linked list of what are called free notes, okay.

And it always keeps the address of the very first free note, and because you're not using this as a client, the heap manager uses it as a variably sized node that – right in the first eight or four bytes – keeps track of how big that node is. Does that make sense to people?

So, it might have subdivided that, and to the left of that line might have the size of that node, and to the right of that line might actually have a pointer to that right there, okay. Now it's not like there's ints and doubles typing any of the material over here. The heap manager has to do this very generically so it constantly is casting addresses to be void*s and void**s, okay, in order to actually jump through what this thing called the free list behind the scenes to figure out which node best accommodates the next malloc request, okay. When you free this node right here it has to be threaded back into the free list, okay. Does that make sense?

I'll be a little bit more detailed come Friday as opposed to this hand wavy after 11:50 comment, okay. But I want you to understand all this. Okay, see you on Friday.

[End of Audio]

Duration: 53 minutes

ProgrammingParadigms-Lecture08

Instructor (Jerry Cain):Hey, everyone. Welcome. We actually have some handouts for you today. We just decided to hand them out after you all sat down. So you'll be getting three handouts and they should be posted to the website, as my TA Dan Wilson is gonna do that sometime before 11:00 a.m. – before noon. Well let's see, last time, I had just given you a little bit of an introduction as to how the heap is managed by software that's included in C libraries that are linked against all of your own code. Every time something like six degrees, or RSG or vector test, or whatever, is created – to remind you from where we were last time, we're gonna start talking about all the various segments in memory. Each application behaves as if it owns all of memory. We'll give more insight into that in a few lectures. I'm specifically interested in the segment that, in practice, is usually – I mean very approximately, but usually right around here. When an application is loaded into memory, the lowest address and the highest address of the heap are advertised to a library of code that's responsible for implementing malloc, free and realloc. Okay? Now this is a general sandbox of bytes. Malloc just hands out addresses to the internals. It actually records, behind the scenes, how big each figure actually is, so that when free is called and passed one of those three pointers there, it knows exactly how much memory to donate back to the heap.

I'll explain how that works, in a second. Because this is managed by software, malloc, free and realloc, the person implementing that can use whatever heuristics they want to make it run as quickly and efficiently and, obviously, as correctly as possible. And so we're gonna talk about some of those heuristics and how that works. Now you've always been under the impression that, when you do this – I'll just call it ARR is equal to malloc – and you do something like 40 times the size of, oops, the size of int. And actually, I'll go ahead and strongly type this pointer, to not be a void star but to be an int. You're under the impression that you get actually 16o bytes back.

I can tell you, right now, that you do not – you certainly have more than 160 bytes set aside on your behalf. If this is the heap – I won't draw all the other nodes that have been handed out, but at some point, it discovers a node that is large enough to accommodate that request right there. We think of it as perfectly sized to be 160 bytes. This address is ostensibly handed back to you and when you pass this either to realloc or to free, it actually can confirm internally – or no, I'm sorry, it can't confirm, but it just assumes that the pointer that's handed to free or realloc is one that has been previously handed back by either a call to malloc or realloc. Okay?

So that is2 some legitimate pointer that has been handed back. The way memory is set aside on your behalf is that it actually allocates more than that number. Okay? For a variety of reasons. But what it will normally do before it hands back an address to you, if you ask for 160 bytes, it'll usually set aside 164 bytes or 168 bytes. Why? Because it'll actually include space at the beginning of either 4 or 8, or actually 16 or 32, whatever it decides, a little header on the full mode, where it can actually lay down a little bit of information about how big the node is. Does that make sense to people?

So if this really is 160 bytes and this is 4, it might, among other things, write down a 164 inside that 4-byte figure. Okay? If it's 8 bytes, it can actually write more than just the size. When you get a pointer back, you actually don't get a pointer to the head of the entire node; you get a pointer that's 4 or 8 bytes inset from the beginning. Does that make sense? You have access, seemingly, to all of that space right there. When you pass the pointer back to free – let's forget about realloc for the minute – for a moment – free says, "Oh, I'm just assuming that this a pointer that I handed back earlier. If I really handed this back, then I know that I put down, as a little footprint, how big this node was. So I'm gonna cast this pointer to be an int star or a long star, or a long long star, or whatever it wants to. So that I can gracefully back up 4 or 8 bytes, interpret those 4 or 8 bytes, in a way that I know I laid down information before." And say, "Oh look, there's a number 164." That means, from this point to that point right there, should somehow be threaded back into the heap. Does that make sense to people?

Okay. Given that right there, here are a couple of problems that I want you to understand why they don't work. Array is equal to malloc 100 times the size of int. So this is a legitimately allocated block of 100 arrays. You know that you're gonna get either 104 bytes or 108 bytes, just a little bit more – I'm sorry 404 or 408 – and then, you go ahead and you populate the array and you realize that you don't need all the space. Maybe you only need to use 60 of the integers and you've recorded that in some other variable. Okay? And so you think you're being a good memory citizen and you do this: ARR plus 60. Now the code wouldn't get written that way, it would probably be framed in terms of some local variable, say "num ints in use" or something like that, or "effective length," and you might because you're more sensitive to memory allocation and you might want to donate back what you're not using, might think that that should work. Well, if this is the 400 byte figure that you've logically gotten, and you've been handed that address and there's a pre-node header of meaningful information there, meaningful to malloc and realloc, and you go ahead and hand back that address to free, depending on the implementation it may be very naïve and just assume, without error-checking, that it was something that was handed back via malloc or realloc before. It might do some internal checking, but malloc and free realloc are supposed to be implemented as quickly as possible and not do the error checking for you because they're assuming that you're really good at this C and C+ + programming thing, So they're not gonna interfere with execution by doing all of this error checking every single time free and malloc get called. So if you were to do this, right here, and it doesn't do any integrity checks on the number that it gets, it will blindly back up 4 or 8 bytes, whatever happens to reside in the two integers, or the one integer at index 58 and 59, would be interpreted as one of these things right here. And if it happens to store in the place where 164 is stored, right there, if it happens to store the number 29,000, it will go from this point forward 29,000 bytes and just blindly follow whatever algorithm it does to integrate a 29,000 byte block back into the heap. Okay?

Does that make sense? Now you're not gonna say, "Sure, what impact that has on the heap and what data structures look like." I'll give you a sense in a second. But the bottom line, the takeaway point here, is that you can't do that and now you have some insight as to why. Okay?

If you do this, int array 100, and you statically allocate your array, and you don't involve the heap at all, and you use it and because you're new to C programming and you think that you have to free the array, if it doesn't do any error-checking on the address, it's not even obligated to do any error-checking to confirm that it's in the heap segment in the first place, it would go to the base address of your static array, back up 4 or 8 bytes, whatever figure and bit pattern happens to reside there would be interpreted as the size of some node, okay, that should be incorporated into the free list data structure. Okay, I'm sorry, I shouldn't say free list because you don't know what that is yet.

Incorporated into the collection of free nodes that the heap can consider for future – in response to future call to malloc and realloc. Okay? Is this sitting well with everybody? Okay? The best implementations – or well, best is up for debate – but the fastest implementation is: Don't do error checking. They'll use heuristics to make sure they run as quickly as possible. Some implementations, I've never seen one, but I've read that some implementations do actually basically keep track of a set of void stars that have been handed back.

And it'll do a very quick check of the void star, that it gets passed to it, against – and make sure it's a member of the set of void stars that have been handed out. And if it's in debug mode, it might give an error if it's not present. If it's not in debug mode, it may ignore it and say, "I'm not going to free this thing because it will only cause problems later on." Okay? Does that make sense to people? Yes, no? Got a nod. Okay.

Now, as far as this 160 is concerned, that is not exactly a – that's not exactly a perfect power of 2. Implementations that I've seen, if this is the entire heap, I've seen this as a heuristic. When you pass in a numbytes figure to malloc, let's say that it's 6, if you quickly recognize whether or not the number that's supplied is less than, say, 2 to the 3rd or 2 to the 5th, or 2 to the 7th, basically categorize and throw it in a size bucket as to whether it's small, medium or large. I've seen some implementations actually divide the heap up, so that anything less than or equal to, say 2 to the 3rd equal to 8 bytes, is allocated from right there. Anything less than or equal to 2 to the 3rd, I'm sorry, 2 to the, like, 6th equal to 64, might be allocated from this segment. And it would always give out a block that is, in fact, exactly 64 bytes or 8 bytes long. Okay? In other words, it won't try to actually perfectly size everything because that takes a lot of work. If it can take some normative approach as to how it clips off individual segments within each sub segment, it might actually have a much easier time keeping everything clean and organized. Okay? As long as it allocates – if you ask for 160 bytes, if it goes ahead and allocates 192 bytes, or 256 bytes, you actually don't mind all that much because at least you're given – you're certainly given enough memory to meet the 160 request. Okay? Does that make sense? Yeah?

Student: That doesn't necessarily mean that, if you have an array of 160 bytes, you [inaudible]

Instructor (Jerry Cain):Right, you're not supposed to rely on implementation details because the implementation of malloc, free, and realloc on, say, you know, one flavor of

Linux may actually be different than it is on another flavor of Linux. I mean, probably not. I'd say all of those compilers are probably GNU backed, so it's like GCC. But like, for instance, Code Warrior vs. X code on the Macintosh. They are implemented mostly, I mean primarily, by two different sets of engineers. And one may use one different heuristic for how to allocate stuff and those who wrote X code may have gone with a different approach.

So you certainly can't assume you have that memory. You're just supposed to assume that you've got the 160 bytes and that was it. Okay? Do you understand, now, a little bit why running over the boundaries of an array can cause problems? Sometimes, it doesn't. The most common overrun, actually, is at the end of the array. So you'll do something like i less than or equal to 10 as opposed to i less than 10. You'll write one space too far. Consider the less common but certainly not unrealistic situation where you actually visit all your elements from top to bottom, you go one element too far, and you actually access and write to array of negative 1. You know where that resides. It happens to overlay the space where malloc and realloc actually put down information about how big the node is. So if you, for whatever reason, go and zero out from 100 down through zero, well then you actually go 1 too far then you zero out the 1 byte – the 4 bytes where malloc is really relying on meaningful information to be preserved. And it will completely toy with the implementation of malloc and realloc in its ability to do the job properly. Okay? Does that make sense? Okay.

What else did I want to talk about? As far as how it keeps track of all of the different portions within the heap, that are available to be handed back to the client, in response to malloc, realloc calls, let me go with this picture. Let's assume that this is the entire heap and I'm writing it to emphasize that it's a stream of bytes. And a snapshot of any one moment – this is allocated out, right here, and this is allocated out, and let's say that this is allocated out, as well. Each of these will probably have a pre-node header at the front. Okay? With meaningful information – they can actually keep more than just the size of the node.

In fact, some implementations – I have seen this, will not only keep track of how big this node is, but it'll actually also keep track of whether or not the node after it is free or not. So it can kind of simplify or optimize the implementation of realloc. So it can keep a pointer to this right here. Does that make sense to people? Okay? And it can go right here and see what – how big it is and whether or not it can accommodate the stretch that is an option in response to a call to realloc. As far as all these blank nodes are concerned, it wants to be able to quickly scan the heap for unused blocks whenever malloc and realloc are called. What will typically happen is that it will interpret these nodes right here as variably sized nodes. You're familiar with that from Assignment Two and it'll overlay a link list of blobs over these 3 ravens right here. So the beginning of the heap is right there, right at the beginning, so it knows where the next pointer is. It'll use the first 4 bytes of the unused blob to store the address of the next blob. Okay? It'll store the address of the next blob right there, and then maybe, it'll put null there or maybe it'll cycle back to the front and use something of a circular link list approach. Every single time you call malloc or free, it obviously wants to traverse this link list and come up with some node that's big enough to meet the request. More often than not, I see a heuristic in place that just, basically, selects the first node that it can find that actually meets the allocation request. So if this type of thing isn't in use and it's not segmented down into sub segments, and it's just one big heap, it might start here. And if it's just looking for a figure big enough to accommodate 8 bytes, maybe this one will work. Okay?

If it needs 64 bytes and this is only 32, but this is 128, it will say, "You're not big enough, but you are." Does that make sense? So it would approach this first fit heuristic in searching from the beginning. There are other heuristics that are in place. Sometimes, they do aggressively search the entire heaps free list. That's what this thing is called. That's what I used earlier. And it'll actually do an exhaustive search because it wants to find the best fit. It wants to find the node that is closest in size to the actual size that's needed by the call to malloc so that as little as memory as possible is left over in some free node. Does that make sense? Okay?

I've seen – I've read about, although I've never seen, that sometimes they use a worst fit strategy. Which means they basically scan the entire heap for the biggest node and use that one with the idea that the part that's left over will be still fairly big. And we're not likely to get little clips of 4 and 8 bytes which are gonna be more or less useless for most malloc calls. Does that make sense?

I have seen heuristics where they actually remember where they left off, at the end of malloc. And the next call to malloc or realloc – I'm sorry, the next call to malloc will actually continue from that point. So that, all parts of the heap are equally visited during an executable that runs for more than a few seconds. Okay? So you don't actually get lots of complexity over here and then this unused heap over here. Okay? Does that sit well with everybody?

There are all types of things that can be done here. There's extra meta-information can be stored here, and there, and there, about what comes afterwards, so it can simplify the implementation of free and realloc. If I go ahead and free this node right here, when it's actually freed and donated back to the heap, none of this is changed, but the first 4 bytes actually set to thread to that right there. And this right here would be updated to point to there instead. Does that make sense to people? Okay? It wouldn't actually go out and clear any information because it just doesn't want to bother doing that. It could, but it's just time consuming. And this could, in theory, be 1 megabyte and why go through and zero out 1 megabyte of information, when the client's not supposed to touch it again? Okay?

Let me just erase this, not because it's would be cleared out, but just so it looks like a free node, like all the other ones. Some implementations would actually prefer not two so-and-so size nodes together but one big node, since they can't see many disadvantages to having one very large node vs. two side-by-side smaller nodes. Does that make sense?

So some of them will go to the effort of actually coalescing nodes so that the free list is simpler and they have a little bit more flexibility as to how they chop things up. Okay?

Make sense? I have seen, and this is kind of funny, I have seen some implementations of free actually just record the address as something that should be freed, ultimately. But it actually doesn't commit to the free call until the very next malloc call or, in some cases, until it actually does need to start donating memory back to the free list because it can't otherwise meet the request of malloc and free alloc – malloc and realloc. Does that make sense?

So I'm speaking in, like, run-on paragraph form here about all the things that can be done. That's because it's written in software and people, those that design these things are typically very good systems programmers. They can adopt whatever heuristic they want to, to make the thing run, certainly correctly but as efficiently and elegantly as possible. Okay? And if, ultimately, this all starts out as one big random blob of bytes, the way the heap is set up is that the address of the entire free list is right there and the first 4 bytes have a null inside of it. Okay?

And that basically means that there's no node following this one. It would also probably have some information about how big the entire heap is. Okay? Because that's just the type of information that's maintained on behalf of all nodes at all times, so it just happens to be the size of the heap when things start out. Okay?

Now, consider this problem right here. Here's the heap again and let's say the entire thing is 200 bytes wide. This is free and it's 40 bytes. This is allocated, and let's say it is 20 bytes, not drawn to scale. Let's say that this is 100 bytes wide and it is free. Let's say that this is in use; it is 40 bytes wide. And is that gonna work out? No, let's make this a little bit smaller. Let's make this 80; make this 20 and so this, right here, is unused. And of course, the heap is much bigger than this and the way it's been chopped down in used block and unused blocks, is a little bit more complicated than this. But you certainly understand what I mean, when I say that there are 160 free bytes in my heap of, that's normally of size 200. Okay? If I go ahead and make a malloc request for 40, it could use this because it's the best fit or the first fit. It could use this because it's the worst fit. Okay? It can use whatever it wants to as long as it gets the job done and you're not blocked by a faulty implementation. If I ask for 100 bytes, it's not gonna work out, right? Because, even though the sum of these free nodes is a whopping 160 bytes, they're not uniformly aggregated in a way that you need when you malloc 100 bytes or 160 bytes. Okay? I mean, you really have to assume these things are gonna be used as an array of ints or structs, or whatever. And you can't expect the client to really absorb a collection of pointers and figure out how to maintain an array or overlay an array over multiple blocks. So you may ask yourself, "Well, can I actually slide this over and slide this over even further, to come up with this type of scenario where this at the front and this is right next to it? And then I really do get 160 bytes that are free – 20 and 20, so that if I ask for 100 bytes, I can actually use it?" This process, it does exist. It won't exist in this problem for – in this scenario for reasons I'll explain in a second. But what I'm really doing here is I'm just basically compacting the heap like a trash compactor compacts trash. Okay? Try and get it as close to the front as possible, okay because you know that, what remains after everything's been sifted to the front, is one very large block. Does that make sense? Okay? That's all fine and dandy except for the fact that you, very likely, I'm sure it's the case, have handed that address and that address out to the client. And it's going to resent it if move the data out of its way. Okay? For you to bring these over here means that the client is still pointing there and right there and now, they have no idea where their data went. Okay? That's a problematic implementation.

So there are some benefits of actually compacting this heap. Okay? Now I haven't seen any modern systems do this, but I know that the Macintosh did this about 12 years ago, when I was doing more systems work on the Macintosh. Oops, you know that's fun. Recognizing that there are some advantages to being able to compact the heap to create fewer large nodes as opposed to lots and little fragmented nodes, they might clip off some part of the heap. This is used in a traditional manner by the heap manager. It's directly dealt with by malloc, free and realloc. But they might clip this part off and, even though it looks small on the board, it would in theory be a pretty large fraction of the entire heap, which is, you know, megabytes or – megabytes or gigabytes in size. It would manage this via handles. So rather than handing out a direct pointer into the heap, like we have up there, and we saw that interfered with heap compaction, what some operating systems did, in addition to malloc and free and realloc, would actually hand out what are called handles, which are not single pointers directly to data, but pointers that are two hops away as opposed to one hop away from the data. And you may say, "Well, why would they want to do that?" Well, they would not only hand out double pointers, but they would actually maintain a list of single pointers. If you ask for 80 bytes via a handle as opposed to a pointer, then it could maintain a table of master pointers and hand out the address of that to the client. Okay? So the client is now two hops away from his data, but the advantage is that this is owned by the heap manager and, if it wants to do compaction on this upper fourth of the picture, it can do it because it can also update these without affecting your void double stars. Does that make sense? Okay?

I saw that used in Mac OS 7.6 and Mac OS, like 8 – not actually 8 never existed, but like all of the 7 series of Macintosh Operating System, probably from 1994 and '95, when I was doing some consulting work that involved this. Very clever idea, the problem is that thing is compacted in the background, at a low-priority thread. Okay? Or it becomes higher priority if there's a demand for this and there's no space for it. You can't actually have the heap compaction going on simultaneously to an actual double d reference request because you can't actually have data sliding in the moment while you're trying to access it.

So the paradigm, or the idiom rather, that people would use for this is, if you really wanted to get memory to something that's compacted and managed more aggressively by the heap manager, you would do something like this. Void star star handle and it would be some function like new handle. That's what it was in Mac OS 7.6 days. You might ask for 40 bytes. Okay? Recognizing that the 40 bytes might be sliding around in a low-priority thread in the background, to keep that thing as compact as possible, you wouldn't bother with it when you knew that you were gonna certainly read to but even read from those 40 bytes. That you actually, somehow, had to tell the operating system to stop moving things around, just long enough for me to read and/or write. Okay? So you would do something like this, handle lock, which basically puts safety pins on all the blocks in that upper fourth of the diagram and you say, "You can move everything else around, but you better not change the pointer that's maintained in this table, that's addressed by my handle. Because I'm gonna be annoyed if you do because I'm about to manipulate it right here." And if you're a good programmer, when you're done, you unlock, you call handle unlock, so that flexibility has been restored for the heap manager to start moving things around, including the block that's addressed right there. Does that make sense? So there's a hint of, like, concurrency there. We'll talk about that more in a few weeks. But that's how heap compaction, as an idea, can be supported by a heap manager, okay, without interfering with your ability to actually get to the data. Okay? You never really lose track of it because you're always two hops away as opposed to one hop away. Okay? Does that make sense? Okay, very good. I'm trying to think what else. I think that's enough. Just kind of a hodgepodge of ideas. I'm not gonna have you implement any of this stuff, okay? But we've actually – I'll try and dig up a section problem, not for this week but for the week after, where people on the mid-term implemented a little miniature version of malloc, where you actually had to understand all of this stuff. Okay, now I can tell you right now that I'm not gonna do that because that problem didn't go very well, when I saw it was given. But it's certainly a suitable section problem and something that you can do in like a less time-pressured environment. Okay? Make sense? Okay. Software managed. It is managed entirely by malloc, realloc and free. What I want to do now, is I want to start talking a little bit about the stack segment. Let me get some clear board space here. The first 20 minutes of this actually very easy to take. It's Monday that'll be a little bit more intense. When I talk about the stacks thing, I drew it last time; it's typically at a higher address space, as a segment. This is the entire, as a rectangle, the entire thing is set aside as a stack segment, but you're not always using all of at any one moment. In fact, you're using very little of it when a program begins because there's so few active functions. Okay? Usually, the portion that's in use is roughly – it's very rough –roughly proportional to the number of active functions, of the stack depth of all the functions that are currently executing. Okay? I will draw more elaborate pictures within this in a little bit. But let me just invent a function and I don't care about code so much as I just care about the local variable set. Let's say that the main function – let's not worry about any local, any parameters – let's say that it declares an int called "a." Let's say that it declares a short array called "b" of length 4, and let's say that it declares a double called "c." Okay? Actually, I don't want to call this "main." Let's keep it nice and simple; let's just call it the "a function." All of the implementation of "a" does, is it calls "b" – I'm not concerned about passing parameters, and I'd call "c" afterwards, and then it returns. We'll blow this off for a second. When you call the "a" function, obviously space needs to be set aside for this int and those 4 shorts and that single double there. 1-byte figure, four 2-byte figures and one 8 byte figure. Not surprisingly, it's somewhat organized in the way that it clips off memory. It doesn't take them from the heap; it actually draws the memory from the stack. All of the memory that's needed for these four variable right here – I'm sorry, three variables and technically, like six because there's four of them right here, they're packed as aggressively as possible and there's even some ordering scheme that's in place. Because "a" is declared first, "a" gets a 4-byte rectangle, and because "b's" declaration is right afterwards, it's allocated right below it in memory. Okay? This being an 8 byte double would have a rectangle that's twice as tall and I'll just do that to emphasize the fact that it's two [inaudible] as opposed to one. Okay? Does that make sense?

When you call this "a" function, what happens is, if I go ahead and just represent – I can't – I don't want to draw this entire thing again. So just let this little picture right here be abbreviated by the hexagon. Okay? This thing is what's called a activation record, or stack frame, okay, for this "a" function. Let's assume that "a" is about to be called. When "a" is called, this pointer, that's internal into the stack segment, it actually separates the space that's in use from the space that's not in use. It'll actually decrement this by size of hexagon. Okay? Do you understand what I mean when I say that?

And that means that the memory right here is interpreted according to this picture, right here. It's like this picture right here overlays those 20 bytes that the stack pointer was just decremented by. Okay? Notice it doesn't lose track of all the previously declared variables that are part of functions above "a" in the stack trays. And when "a" calls "b," and calls "c," it'll decrement the stack pointer even a little more. Does that make sense? Okay. Maybe element "b" – I'll keep these simple. Void "b" declares an int called "x," a car star called "y" and let's say a car star array called "z" of length 2. The stack frame would look like this. "X" would be above "y's" 4 by pointer, and then, this is an array of size 2. So this is a total of 16 bytes, there. I'll abbreviate it with a triangle. Okay? And let's say that "b" actually calls "c" as well and "c" has a very simple local variable set, I'll say a double m of length 3, and I'll just do a stand-alone int called "m" and it doesn't do anything that involves any other function calls. The activation for this would be big, there's three doubles; there's a stand-alone int below it; there's "m" – this is all of "m" of zero through 2. And I'm gonna abbreviate this with a circle. Okay?

So the very first time "a" gets called, you know just the work flow will bring you from "a" to "b" to "c", back to "b", back to "a", which will then call "c". Okay? So as far as the stack is concerned – let me just draw a really narrow stack, okay, to emphasize the fact that the full width is being used to help accommodate the stack frames right here, when "a" is called, whatever's above here, we have no idea. They're obviously gonna correspond to local variables; they're preserving their values; there's just no easy way to access them, unless you happen to have pointers passing those parameters. Okay?

When you call "a," this thing is decremented by size of hexagon. Oops. I needed to remind myself what a hexagon looked like. That's that and then, this pointer right here, is kind of the base or the entry point that grants the "a" function access to all of its local variables. It knows that the last parameter "c" is at an offset of zero from this pointer right here. Okay. 8 bytes above that is where the second to the last variable that was declared can be accessed, etc.

This pointer is actually stored on the hardware; we'll see that on Monday. But this actually what's called the stack pointer and it always keeps track and points to the most recently called functions activation record. When "a" calls "b," it decrements this even further. I'm gonna stop drawing the arrow and the triangle activation record is laid right below that. Okay? Access to this, it just isn't available to the "b" function because "b" doesn't even know about it. It just – it's incidental that it happens to be above it, but "b" can certainly access all the variables that part of the triangle activation record. Since "b" calls "c," it's gonna lay a circle below that. When it calls "c," that circle will be alive and in use, the stack pointer will be pointing right there until "c" is done. When "c" exits, it simply raises the stack pointer back to where it was before "c" was called. Whatever information has been written there, okay, it actually stays there. Okay? But it's supposed to be out of view; it can't be accessed. Okay? So – I'm sorry, "c" returns and then "b" returns and we come back to that situation and the drawing there technically corresponds to the moment – we've just returned from "b" but we have yet to call "c." Right? And so when it calls "c," it follows the same formula. It has no idea that "b" has been called recently. And the formula here is that it just overlays a circle over the space where the triangle was before. Okay? So it actually layers over whatever information was legally interpreted and owned by the "b" function. But it doesn't know that and it's not really gonna be the case that "c" has any idea how to interpret "b's" data. It doesn't even necessarily know that "b" was called recently. All it's really doing is it's inheriting a random select – a random set of bits that it's supposed to be initialize to be meaningful if it's going to do anything useful with them. Okay? Does that make sense? Do you now see why this thing is called a stack? I'm assuming you do. Okay? Every time – before you can actually access "b's" variables, if you call "c" – "c" has to be popped off the stack the stack frame, before you can come back to "b." Okay? Now we're gonna be a less shape-oriented in a little bit and start talking about assembly code and how it manipulates the stack. So we're gonna suspend our discussion of the stack for, like half a lecture, okay? But now, I want to start focusing a little bit on assembly code. Okay? And how that actually guides execution. We're ultimately going to be hand compiling little snippets of C and C + + code to a mock assembly language. Nothing – it's not an industrial thing like MIPS or X86 or something like that. That would be more syntax than concepts. So we have this very simple syntax for emulating the ideas of an assembly language. And I want to start talking about that now. There is a segment down here. It usually doesn't have to be that big because the heap and the stack – the stack is actually never this big when it starts out; it can be expanded. The heap is usually set aside to be very big because anything that's– any huge memory resources are typically dynamically allocated while the program is running. This right here, I'm gonna refer to as the "code segment." Just like the heap and the stack, it stores a pattern of zeros and ones there, but all of the information in the code segment corresponds to the assembly code that compiled from your C and C + + programs. Does that make sense? Okay. So I'll give you an idea of what these things look like in a little bit. Let me just talk about, at least at the cs107 level, what a computer processor looks like. So I've already drawn all of RAM a couple of times, I'll do it one more time. There it is; I don't have to break it down, just stack and heap and code. That's all of RAM. In any modern computer, there's usually – now it's very often that there's more than one of these things, but we're just gonna assume a UNA processor, okay, where this is relatively slow memory. You're not used to hearing of RAM, which this is, as slow. But it's slow compared to the memory that's set aside in the form of a register set. I'm drawing it – it makes it look like it's one-fourth the size of RAM; it's not; it's much smaller. In our world, we're gonna actually assume that all processors have 32 of these registers, where a register is just a general purpose 4 byte figure that we happen to have really, really fast access to. Okay? Does that make sense to people? I'm going to call this R 1; I'm going to call this R 2; I'm going to call this R 3 and I'm going to draw a 3 instead of a 2, etc. And those are going to be the names for these registers. The registers, themselves, are electronically in touch with all of RAM. There's a reason for that. Okay? And when I'm doing this, I'm just drawing this big network of silicone wires that are laid down microscopically – okay, not microscopically, but you know what I mean. So that, every single register can technically, draw and flush information to and from to and from RAM. Okay? The 16 or the 32 registers right here, are also electronically in touch with the electronics. It's always drawn this way; I'm not really sure why. What's called the "arithmetic logic unit" or the ALU, that is the piece of electronics that's responsible for emulating what we understand to be addition and multiplication, and left shifting and right shifting of bits, and masking it. All of the things that can be done very easily on 4 byte figures. Okay? Does that make sense to people? So we can support plus and minus, and times, and div, and mod and double less than and double greater than, and double ampersand and double vertical bar, and all of these things because of this ALU right here. Okay? It is electronically in touch with the register set. There are some architectures that use a slightly scheme right here. I'm going with an architecture or an assembly code – I'm sorry, I'm going with a processor architecture where the ALU is just in touch with these things right here, and it's not directly in touch with general RAM. The implications of that is that all meaningful mathematical operations have to actually be done using registers. Does that make sense?

Now you may think that that's kind of a liability, that you'd have to actually take something from memory, load it into a register, in order to 1 to it. Okay? Well, that actually is what happens. The alternative is for you to get this right here to electronically in touch with all of RAM and that would be either prohibitively expensive, or it would be prohibitively slow. Okay? So they can optimize on just load and store between registers and general RAM and then, once they get something into the register set, that's where they do all the interesting stuff.

So the typical idiom that is followed for most statements – anything mathematical – think i plus plus, or i plus 10 or something like that, is to load the variables from wherever they are, either in the stack or the heap, okay; load them into registers; do the mathematics; put the result in some other register, and then flush the result that's stored in the register, out to where it really belongs in memory. Okay?

Just assume, without committing to assembly code right here, that that's the space that happens to be set aside for your favorite variable, i. And it has a 7 inside of it. Okay? Let's say that somewhere, probably close by, there is a 10. Okay? And you're curious as to what happens on your behalf at the memory level in response to that right there. Well, truly, what happens is the j plus i compiles to assembly code and the assembly code is the recipe that knows how to load j and i into register set; do the addition and then flush the result back out to the same space that j occupies. Okay?

But just in terms of actually seeing where – how the 7 and 10 move around, what would probably happen is the 7 would be loaded into R 1. The 10 would be loaded into R 2. You can actually add the 10 and the 7 and store the result in R 3 because these two registers are in touch with the electronics that are capable of doing that addition for you. And after you synthesize the result right here, you can flush it back out to j. So given what I've shown you, so far, it's not a useless metaphor for you to just think about assembly code instructions as byte shovelers, okay, to and from RAM, and also doing very atomic, simple mathematics. Okay? Plus, minus, things like that. Okay? The components of assembly code that really are different are usually implementation. I think the double e aspects of computer architecture are very difficult, probably because I've never really studied it; I'm a compute scientist. Okay? But also, the parts that are interesting to us, is how something that is stated, usually quite clearly, in C or C + + code, actually gets translated to all of these assembly code instructions, such that the assembly code, which is in touch with memory, actually executes and imitates the functionality of the C and C + + code. Okay? If this is my C + + code, okay, then I need this to become a 17. How does the assembly code actually do that for me? How do I have this one to one translation or one to many translation, between individual C and C + + statements, okay, and the assembly code instructions that have to be done in sequence in order to emulate that, right there. Does that make sense? Now, given these pictures over here, you shouldn't be surprised that j doesn't move as a variable while that program – while that statement is running. Okay? So it's always gonna be associated with the same address in RAM. Okay? That's good. You don't want it flying around when you try to write a 17 back to it. Okay? You may ask why I don't – why they wouldn't – what are the disadvantages of trying to get this to be in touch with general memory? I mean, if we have to have this in touch with that – if we have to have this complex matrix between the register set and this right here, okay, then why not just have it this complex matrix of wires between general RAM and the ALU? It just makes the implementation of the hardware that much more complicated. There's no way of getting around at least a few set of operations between RAM and the register set. Okay? At the very least, you have to have load and store instructions to move four 2 and 1 byte quantities to and from RAM, in between the register set. Does that make sense? You have to have at least those. If you actually try to support addition between two arbitrary addresses, it can technically be done. It might make a clock cycle that's actually on the order of seconds. Okay? But it technically can be done. But hardware designers obviously want the hard – want to be able to do any form of atomic operation at the hardware level, but they also want it to run as quickly as possible. So given that this right here would have to correspond to more than a few actions; load i into register; load j into a register; do the addition and write things back. There were, like basically four verbs in that sentence, four actions that needed to happen in order for that to emulate that right there. What I'm loosely saying is this is a statement that will correspond or gen – will – this will compile to four assembly code instructions, two loads, an addition and a store. If it tried to optimize and do all of this in one assembly code instruction, it could do it. Okay? It would actually probably require that this be in touch with that right there, but the clock cycle would have to increase because whenever you have a more complicated hardware implementation, it's generally the case that the clock cycle speed goes up. It may go up – if it goes up by more than a factor of 4, then you'd actually prefer the really simple, load; do the work here and then flush out. Because in the end, you're really worried about correctness but also speed. Okay? And if you have this very robust implementation, where everything can be done directly in memory, but the clock cycle is on the order of, like, milliseconds as opposed to nanoseconds – or not nanoseconds, microseconds, okay, then you actually prefer to go with the simpler idea, this load store architecture, where you always load things into registers, manipulate them in a meaningful, mathematical way and then flush the result out to memory. Okay? So I actually don't have any time. I have 25 seconds; I can't do very much, then. What I will do on Monday is I will start seemingly inventing assembly code instructions. And I will come up with syntax for actually loading into a register a 4-byte figure from general memory, doing the opposite, taking something that's in a register and then flushing it out. And then, talking about what forms of addition and subtraction and multiplication are supported between two different registers. Okay? We'll get –

[End of Audio]

Duration: 50 minutes

ProgrammingParadigms-Lecture09

Instructor (Jerry Cain):Hey, everyone, we're online. I don't know have any handouts for you today. I started to introduce the next segment of the course as being the part where we actually cover computer architecture and assembly language. We're gonna spend a lot of time trying to figure out exactly what our C and C++ code snippets, or just the C and C++ code, although we'll only deal with snippets in any one example, how it actually compiles to assembly code. Talk a little bit about the memory the models, talk a little bit about how function call and return works, and also to expose you to, not a real assembly language, but at least a little mock one that we've invented for CSten7 purposes to kind of show you all of the little gears that are in place to get addition and multiplication and division and function call and return and pointers, and all those things to actually work, but at the hardware level. Okay? Now, when I left you last time I had started talking about the stack segment. And if you remember, probably about halfway through Friday's lecture I had hexagons and circles and triangles as placeholders for basically the skeletal figures that overlaid memory, and showed you how all the local variables in a particular function were actually packed together in what was called an activation record. In many ways, the assembly code that we're going to be writing, are really just these 4-byte instructions. They're ultimately 0's and 1's, but they're interpreted by the hardware to access variables within the triangles and the hexagons, okay, and pull them into registers. Maybe add one to them, or add ten to them, or pass it to some helper functions just to get the assembly code to imitate exactly what your C and C++ code was written to do. Okay?

So here's the stack segment. Let me just contrive this one little block of code. All right, int i, I have int j, and then I do i is equal to ten. And then I do j is equal to i plus seven, and then I'll do j plus plus. Let's just assume, not surprisingly, that i and j are packed together as some 8-byte activation record and they reside somewhere in the stack segment. I'm just gonna draw it randomly right here. Okay? And that address right there, I'm going to assume, is stored in one of those 32 registers that I named last time; I'm just going to go with the first one and call it R1. So R1 is a general purpose 4-byte bit pattern storer that actually happens to store the base address of the activation record that makes up this code snippet. Okay? Now, you ultimately know that a ten has to be placed there, and that an 18 will eventually go there. Okay? Does that make sense to people, I'm assuming? Okay. But I actually am more focused not on the numbers themselves, as the assembly code that actually does that relative to this address right here. So I'm going to make some assumptions that the base address of the two variables that are relative here, are stored in a special dedicated register. I'm going to call it R1 now; I'll change the name in a little bit. And I'm just gonna illustrate by example what assembly code instructions look like to actually put a ten right there, and then to pull it into a register, add seven to it and flush it back out to the space for j. Okay?

In order to get a ten into that space right there, we don't actually deal with that address specifically, we just deal with an address that's at an offset of positive four from the address stored right there. Okay? The notation for doing this, this one line in our little mock assembly language would compile to that. And that's our first assembly code instruction. Okay? That capital M, it's like it's the name of all of RAM. You think about all of RAM as this big array of bytes, M is the name of that thing, R1 is the base address, four is an offset. This right there identifies the base address of the 4-byte figure in all RAM that should get a ten. Does that make sense to people? Okay. So this is an example of a store operation. And it's called a store operation because it actually updates some region in a stack segment with some new value. The very next few instructions are in place to actually take care of j is equal to i plus seven. I don't want to bank on the fact – or rely on the fact that I know because of this small example that i is equal to ten. I wanna do the most robust thing, which is to actually go and fetch the value of i, pull it into a register where you're allowed to do addition, added a seven to it, and then flush the result out to M of R1. Does that set well with everybody? Okay.

So don't bank on the fact that there happens to be a ten right there. An optimizing compiler might take an advantage of that, but we're not writing an optimizing compiler, we're just trying to brute force without the translation process right here. I wanna do this: into R2, another general purpose register, I wanna load the 4-byte bit pattern that happens to reside right here. Now, this is an example of a load operation. Okay. Let's make it so that that really looks like a load. R2 contains the 4-byte bit pattern for what's residing in i right there, so this corresponds to the understanding that we're going to operate on i's value. Into R3, another general purpose register, I'm going to take whatever R2 has as a bit pattern and add the immediate constant seven to it; this is an ALU operation. It is actually very often the case, though, on the right hand side of some ALU operation, there are either two registers, or there's a single register and a constant, and that wherever the result is to be stored is identified on the left hand side. Okay. That probably makes sense to people. This is a bit pattern, is the thing that has to be replicated in the space that's really set aside for j, and we know where j is because we have R1 storing the base address of it. So right here, I have yet another store operation. So two of the registers, R2 and R3 were piecemeal updated with a ten right there, a 17 right there, and then a 17 is flushed right there. Okay. Does that make sense to people?

This load ALU store sequence is actually very common, and exists more or less on behalf of any kind of assignment-oriented statement in C or C++. Okay? The very next line really expands to j is equal to j plus one, so it has a structure that's very similar to this. I can reuse registers. I can even discard a temporary value that's in R2 if, for whatever reason I want to. I don't have to write to R3 right there, and then I can do M of R1 is equal to the new value that ended up in R2. Okay? I didn't have to be that efficient about the user registers, but it's not a problem if I don't need the old value of j and I just want to deal with the incremented value. It's fine to overwrite R2, and to override its old value. So this has the same exact structure as that right there, it just happens to be dealing with variable in RAM as opposed to two, and this is how this 17 goes from an 18, and R2 would also have an 18 inside of it. So a few things to say about this, by default all of the load and store and ALU operations deal with 4-byte quantities. Okay? It's clearly the case that pointers and ints are probably by far the most common atomic type you deal with in programming, at least in C and C++, so the hardware is set up to optimize, by default, dealing with 4-byte figures. Okay. That doesn't mean we can't deal with isolated characters and shorts, or that we can't deal with doubles and structs as a whole. But I'm more interested in being 4-byte oriented with all of my instructions here. Okay.

A lot of you may be questioning why I bother to do this. You could say, "Well, why don't I just set R3 equal to ten plus seven?" I want to – every single one of these blocks – this corresponds to that statement, these three correspond to that statement, these three correspond to that statement. I want every single block of assembly code instructions to be emitted or rendered in this context insensitive manner. And the reason I'm saying that is because this code is going to work out and do the right thing even if I change this line right here. Do you understand what I mean when I say that? Okay. If I were to hard code in an M of R1 is equal to 17, then that would stop being correct if I changed this line, which was supposed to be completely unrelated to it – to, like, a 12 or a 100. Okay? Does that set well with everybody? Okay. The same thing with this. You could argue, "Well, why don't I just go in and do M of R1 ++?" R assembly code instruction doesn't allow ALU-like operations to be performed on arbitrary memory addresses. You always have to go through the load, do an ALU operation on just registered values, okay, and then flush the result back out to memory. It more or less makes for a simpler assembly language, and it also makes for a faster clock speed when things are that simple. Okay. Make sense to people?

Let's deal with an example that actually doesn't always deal with 4-byte quantities. Let me go ahead and do int i car ch – actually, I don't want to do that – [inaudible]. Let's do short S1 short S2. Okay. The way I've declared these, we would be dealing with an activation record that looked like this: i's right there; this is where S1 is gonna be declared, this is where S2 is gonna be declared. Okay? That looks a little weird, but S2 is the last of the three declarations, and so it's gonna be at the lowest base address according to my model. Okay. I'm always gonna give you variable declarations that work out nicely and that are an even multiple of 4-bytes so that the pictures are always fully rectangular and with no, like, chunks or nips out of the corners. If I do this, that's more or less the same as that instruction up there, with a different number. This would translate to M of R1 plus four, assuming that the register R1 has been set to point to that right there. R1 plus four, memory [inaudible] reference is equal to 200. Okay. That means as a 4-byte figure, the bit pattern for 200 is laid down. Now, 200 is a pretty small number, so in a big ending world, we'd expect one byte of zero followed by another byte of zero, followed by a third byte of zero, followed by one byte that happens to have all of the 200 in it. Does that set well with everybody? Okay. This line has to somehow update S1 – let's actually write it, to be equal to, logically, 200. But it's only supposed to update two bytes of memory. Does that make sense? Well, if you wrote this, your heart would be in the right place, but there are two fairly relevant errors going on here. First of all, you can't do a load and a store in one single operation. Okay. Because that would require assembly code instructions to somehow encode in four bytes the source, memory address, and the destination memory address, and that's difficult to do. So what we wanna do is we want to evaluate i as a standalone expression, and just do this: R2 is equal to M of R1 plus four, again I'm being context insensitive about it.

That means that R2 is gonna have that as a bit pattern inside of it. And then what I want to happen is I somehow want to update this two bytes right here with the 200, but I only want those two bytes to be copied. That's consistent with what we know happens just at the C and C++ level. Okay. Now, if I do this, that won't do what we want because the assembly code that I'm writing right there has absolutely no memory as it's executing what C or C++ code was in place to generate it. And the way I've written it right there, it's just like so many other operations I've put up here. This would be taken as an instruction to update the four bytes from this address through this address, plus three, okay, with new information. And it would update that and that and that and that with four new bytes. Does that make sense to people? I don't want that to happen. This instruction right here would put a zero and a zero and a zero and a 200 right there, and then all of a sudden, S1 would be initialized to zero, and i would become some very, very large number. Okay? That's because I'm mistakenly dealing with a 4-byte byte transfer here and I don't want that. In our world, you override the 4-byte rule by actually putting a little dot-2 right there. It's as if all of these other instructions have an implicit dot-4, but we don't bother writing dot-4 because dot-4 is just for the default. But when you only wanna move around a single half-word, which is two bytes, or a single byte, you'd put dot-2 or dot-1 there. Okay? That's an instruction that when we're sourcing from a register, so just take the lower half of it and update the two bytes that they give in this address right here. Okay? And update this with zero and 200, and that's how S1 becomes 200, and that's how i is left alone as the 200 is a 4-byte quantity. Okay? Does that make sense to people? Okay. If I do this, S2 is equal to S1 plus one, it's very similar to the j++ up there, but we have a lot of dot-2's that are in place to make sure that only two bytes are moved around at any one moment.

This right here would have to load S1 into a register. This is how I would do that: R2 is equal to M – whoops – M of R1 plus two. But the way I've written it right there, it will copy four bytes as opposed to two bytes, unless I do this. What that does is forget about the old value and set R2. When it pulls two bytes into a register, it lays these two bytes in the lower two bytes of the entire register, and then it sign extends it just like it would in C and C++. So it's padded with two bytes of zero's right there. Okay. Now this plus one is just traditional plus one. R3 is equal to R2 plus one; that's what takes this to be 200 to 201. And then when I flush the result out to this other space that has just two bytes associated with it, I do it this way: M of R1 equals dot-2, the result that's stored in R3. Okay. So I understand that these examples aren't all that riveting from an algorithmic standpoint, I'm really just trying to illustrate our assembly code languages as operations on general memory where the general memory is framed as an array of bytes, but everything is taken in 4-byte chunks, by default anyway. Okay. Yep, go ahead.

Student:

Is there a reason why you used a third register in that last –

Instructor (Jerry Cain):

As opposed to there, no.

Student:

– second one?

Instructor (Jerry Cain):I actually – I'm inconsistent with my use of registers, in terms of, like, the conserving. I just conserve them if I anticipate having a lot of them in the example. I just did it there, I should have put R2. I thought of that as I was writing it. Okay. Any other questions at all? Yeah?

Student:

What is the [inaudible]?

Instructor (Jerry Cain):M of R1 identifies this space right there; that corresponds to S2. Okay. So that's receiving some 2-byte bit pattern because of that dot-2 right there. Does that make sense? R3 stores the incremented value, or the result of the plus one operation that's right there. So a full evaluation of the right-hand side ultimately made it into R3. I only have room for the bottom two bytes, which is why I have that dot-2 right there. Okay. So this is how I get zero, 201 right there. Okay.

Now there are other things that go on, but as far as – interestingly enough, except for function call and return, you've seen almost all the mechanics of the assembly code language that I want to teach you. Okay.

Let me do this, let me deal with an array of length ten – no, actually that's too big. An array of length four is big enough, and then I have a standalone integer. And I just wanna go ahead and figure out what this four loop will translate. And with each iteration, I just wanna update some value in the array to be equal to zero. After it's all over I wanna set i minus minus.

Just to say something silly right here, do you understand how this code is simple enough that it's just executed sequentially? The assembly code is executed in sequence as well. First clock cycle it updates memory according to that rule right there; next clock cycle it does that and then that and then that and that and that and that. And over the accumulation of six clock cycles, we've effectively realized these three C statements. Memory is updated in a way that's consistent with the way these three lines have been written. Okay.

There's some looping going on there, clearly. If there's looping in a language, then you shouldn't be surprised that there are at least directives in it in the assembly code language to jump back an arbitrary distance to start over some loop again. Okay. Or with the case of, like, if statements and switch statements; they're based on the result of some test, you actually jump forward four instructions or 12 instructions to the point it starts executing the else clause or some particular case statement. Okay. Does that make sense to everybody? Okay.

So as I write this, I should write assembly code, and you're familiar with how that and that would be executed, maybe to some degree that, as well, although you haven't seen a raise in the context of assembly language, but the test is new to us. The i minus minus is not, but the actual looping certainly is.

You know how there are six different relational operators that can be set in between any two integers? Less than, less than or equal to, greater than, greater than or equal to, double equals, not equals to. It's typically the case in most assembly languages that they have branch instructions that are guided by the same six types of relational operators. Okay.

Let me just write the code for this. Let me draw the picture; the way this would be laid out is that I'd have 16 bytes with four more bytes below it, that's I, this is array of zero through three, but I'll emphasize that this is the zero, the oneth, the twoth, and the threeth. Okay. And assume that R1 points to the base address of the entire figure. The actual hardware will make sure that the base address of the currently relevant activation record – that that address is stored in some register; I just happen to be calling it R1 at the moment. Okay.

So we come here. We assume that R1, as a register, stores the address of that picture up there. And the first thing that happens, is that this thing gets executed exactly once because it's in the [inaudible] portion of the four-loop, obviously. So what happens is that right up front, M of R1 is set equal to zero. And that gets a zero right there. Okay.

We next execute code on behalf of this to decide whether we're going inside the body of the four loop, or we're circumventing it because the test failed. Okay. These are the assembly code instructions that would be expanded on behalf of this test right there. I'll put a double line right there to mean that there's some new C statement that we're dealing with.

I would have to load into R2 the value that's at M of R1. Okay. Again, I'm starting to generate code in a context insensitive manner, I don't want to assume that there's a zero in there. In fact, you'll see in a little bit that there won't always be a zero in M of R1. So I wanna load that into a register, and then I have this branch instruction, b, and I'm gonna leave a blank and a blank right there. We're gonna fill this in with some abbreviation for not equals or greater than or equal to or whatever. We'll decide what it is in a second. It takes three arguments. It takes – whoops – the first register in the comparison or constant, the second register in the comparison or a constant, in this case it's four; and then it takes as a third argument what's called a target address, the place to jump to if the branch instruction passes. Okay. Now we jump forward several instructions – I'm sorry, let's say one C instruction here if this test fails. But if I take the logical and inversion of this, and I jump forward when this test passes, okay, then I'm circumventing the four loop. Does that make sense to people? This is where code for the array of i equals zero will be placed. When this test fails, this test needs to pass so that I jump forward a certain number of assembly code instructions to the part that actually executes that right there. Okay. That means that I want to branch, I want to circumvent the normal pattern of advancing to the next assembly code instruction if this as a number is greater than or equal to four. Okay? And so the abbreviations for these branch instructions shouldn't surprise you; they are: branch on equal, branch on not equal, branch on less than, branch on less than or equal to, branch on greater than, branch on greater than or equal to. Okay. So if R2, which stores the current value of i is greater than or equal to four, we know the loop is over, so we wanna jump forward as if the loop never existed before. Okay. We have to fill this part in. All I can tell you right now is that the address here is gonna be framed as some offset relative to a special register I call PC. Now, PC is really gonna be like the 27th or the 29th or the 31st of the 32 registers, we just give a better name for it. PC stands for program counter, and it stores the assembly code instruction of the currently executing instruction. Okay. I'm sorry, it stores the address of the currently executing instruction. We never know what PC really is, but if this is PC, this is PC plus four, PC plus eight, PC plus 12, PC plus 16, et cetera. Does that set well with everybody? And with each clock cycle, as part of each clock cycle, it by default just updates PC to be four larger than what it was before because all of our assembly code instructions are 4-bytes wide. Okay. And by default, it always just advances to the next one unless some jump instruction or some branch instruction like that tells us to do otherwise. Okay. I'm leaving a question mark right here because I just don't know how many lines array of i equals zero is gonna translate to. Okay. All I can tell you right now is that the offset is gonna be positive, and that it's going to be some multiple of four. So something like plus 16 or plus 12 or plus 24, we have no idea what yet. Okay.

If this branch instruction fails, it's because this test passed which means I'd have to just fall right to the next line of C code, which means I'd have to fall to the next line of assembly code, which implements this right here. Okay. Now you know enough about pointer math, this is easy pointer math. But this has to translate at the assembly code level to something that finds the address of the ith figure in the array and writes a zero there. There's implicit scaling of i times size of integer here; does that make sense? The scaling over here has to be explicit. Okay. If you just write assembly code by hand, which you can do, and you're just really thinking in C and C++ terms while you're doing it, you have to make sure that you assign to an offset of plus zero or plus four or plus eight relative to the base address of the array when you're writing a zero. If you're trying to emulate a four loop inside assembly code, then you have to make sure you take care of the pointer math explicitly. So what I wanna do is I want to reload the value of I because I am being context insensitive about how I use variable values. It's a zero, a one, a two, or a three; we know that. But then what I wanna do, is I wanna take R3 and I wanna multiply it by four. Now that four is there because ints are actually four bytes. I can't write size of int here like I encourage you to, not in this example but whenever you have to manually deal with type sizes in C and C++ code because this has nothing to do ultimately with C or C++. Okay. It happens to be imitating the execution of a C++ program, but all size and type information at the assembly code level is completely gone. It just has to be the case that the code is written in a way that's consistent with this is intended to do. Okay. R3 has the value of i, R4 has the value of i scaled by four, so now R4 has the distance in terms of bytes from the base address of array to where the zero has to be placed. Does that make sense? What is the value of array? We know that it's ampersand of array of zero, right? Array of zero resides right here. Okay. So the ampersand of array of zero, which is synonymous with just plain old array, it is synonymous with R1 plus 4. So I'm gonna do this. This stores the offset; this stores the base address of the entire array. R6 is equal to R4 plus R5 – whoops – R6 has the address within the array that should get a zero on this particular iteration. Okay.

Every single iteration – R5 gets the same value every single time, but R4 certainly does not. So that means that M of R6 is equal to – I'm sorry, M of R1 – oh, no, I'm sorry, that's right. R6 is equal to zero; that's the base address of the place in the stack activation record, that should get the zero and that's why a zero was there. These five lines right there, it's complicated, but they actually are in place to emulate that line right there. If we were writing a CS107 assembly language compiler, this line would translate to these five lines or something that's equivalent to it. Okay. Then what happens next unconditionally, is that we jump back up here to i plus plus, and we execute this. That just does this – R2 is equal to M of R1; R2 is equal to R2 plus 1; M of R1 is equal to R2. And then we know that we execute the test again. So what happens here? Rather than actually writing the code for the test again, you go back and you reuse the same code you wrote for it the first time. So this is an unconditional branch. Okay. We don't use the word branch, we just use jmp because they didn't have room for the u – jmp, and you actually jump to a hard-coded address, but we always frame it in terms of the current PC value. So it's not PC itself, it's PC minus some value. I wanna jump back to this line right here; the line that loads i, compares it to four and decides what it's gonna do. Okay? This minus has to be scaled by four, but I wanna jump back one, two, three, four, five, six, seven, eight, nine, ten instructions. Does that make sense to people? So this would be a PC minus 40. Okay. Right here, I'm out of room, but this is where I would continue. This is where the code for i minus minus would go, and it would be assembly code that's emitted for i minus minus that has nothing to do with this four loop. And it doesn't bank on the fact or the understanding that I would be equal to four at the time it gets there, it would just do the R2 is equal to M of R1; R2 is equal to R2 minus one, and then flush it back out to i. The reason I'm writing that there and the reason I have this here is because I wanna make it clear that this is the place where we should be jumping forward to at the assembly code level when that test passes. Okay? When this test fails, I just do the implicit update of PC to PC plus four. You don't have to write anything for that that just happens by default. If you wanna override what gets used as the new PC value, you have to set PC right here to be PC plus one, two, three, four, five, six, seven, eight, nine, ten instructions. Okay. Times four, so this question mark would become a 40. Okay? Do you guys understand why those numbers are 40?

You may think that this minus value right there and that plus value there have to be exactly the same, that's not always the case. Okay. It just depends on how complicated the test is, but sometimes this right here, which is the part that evaluates the i, this could be something that's arbitrarily complicated, like i plus 24 plus j or something like that. So it might actually be a lot of code that would have to be accounted for in this jump right here. Okay. But this plus 40 deals with an offset from this to the line after the unconditional jump back; it may be a smaller value. Okay. So they're not always the same number. Okay. I think you understand, even if the assembly code is weird for you at the moment, you understand that it really is this brute force translation of this right here to code in a language that just thinks about moving 4-byte quantities around by default. Okay. Short action branch instructions, it has some unconditional branches and some conditional branches. There are gonna be a few more things in place, but really you've seen like 70 percent of the language already. Okay. You guys get what's going on here? Okay. Very good. Questions? Yeah, right there.

Student: If you wanted to, instead of jumping back ten lines, could you just jump back nine because the top line is R2 equals MR1 and you [inaudible] right before the jump set MR1 equals R2?

Instructor (Jerry Cain):Yu could. It actually – it would certainly be correct in the sense that it would do the right thing. That would just be taking advantage of the fact that the register happens to have the right value, but I just wanna be consistent. I'm not going to enforce this – I'm not gonna police the matter to the point where I actually yell at you about it, but I'd actually like you to just get in the habit of generating code in this context insensitive manner. And this load right here, this one happened to do with the fact that there's an i present in the test; does that make sense? This i – I'm sorry, this R2 being set to that right there, okay, just happens to be associated with this i plus plus right here. And as it turns out, actually – I'm sorry, this right here has the right value. It's – I don't want to say it's a coincidence, because it's not a coincidence, it's a four loop and it's the traditional idiom with the four loop. But I'd rather you just be fastidious about just generating it and, like, basically going brain dead about what you've generated code for before because then you know you're always right, okay, regardless of whether or not this changes.

You want the code that's emitted on behalf of this right here and that right there to be the same every single time, or at least allow it to work if it's the same every single time, regardless of what you do right here. Okay. Yeah?

Student: If all of the translations of every single translation translates into 32 bits –

Instructor (Jerry Cain):That's right. We're dealing with assembly code language where all instructions are four bytes wide. Okay. Thirty-two bits, all of them, yeah. Okay.?

Does that make sense to people? Okay. Let me explain a little bit how 32 bits are usually just enough for you to encode an instruction, just to understand what encoding is like. When a 4-byte figure is understood to be an assembly code instruction, it's typically subdivided into little packets or little sub-packets. Here is a 4-byte instruction, and I'll just draw very loose boundaries here for the bytes. We're used to instructions like this: R1 is equal to M of R2 – I'm sorry, M of R2 plus four. Something like R1 is equal to a constant is not unusual either. Maybe you see something like this: R3 is equal to R6 times R10. And then something like M of R1 minus 20 is equal to let's say R19. A load, a direct immediate constant load, an ALU operation, and a store; these are little bit more elaborate than you'd see in practice, but nonetheless we have to be able to support them.

This type of instruction and that and that, they're certainly different from one another. This is a load, this is an ALU operation, this is a store. You understand what I mean when I say that. I even argue that this right here is technically a different type of instruction than this one because this is framed in terms of a register and an arbitrary memory address, and this is framed in terms of the constant. Does that make sense?

Let's say that I've decided that my assembly code language has let's say – let me – 59 different types of instructions. I have to somehow encode in this four bytes right here, which of the 59 instructions we're actually dealing with. Does that make sense? Well, 59, unfortunately, isn't a perfect power of two, so if I really want to be able to distinguish between 59 patterns, I have to be able to distinguish between, it turns out, 64 different patterns, okay? So I might set aside the first six bits of all 32 bits of – of all 32 of them to be the part that the hardware looks at to figure out what type of instructions should be executing. Maybe it's the case that this corresponds to – in this space right here would be called an operation code, or an op code; maybe it's the case that all zeros means this type of load instruction. Maybe this right here is the op code for an immediate constant load into a register. Maybe this right here, being a multiplication, corresponds to that type right there, and then that right there might be let's say all ones. Does that make sense to people?

When the hardware looks at an assembly code instruction, it doesn't actually see these. This is an assembly language instruction that makes sense to us, but it's actually expressed in the hardware as a machine code, which is 32 zeros and ones. It would actually have to look at the first six during the first few percents of the clock cycle, okay, to figure out how to interpret the remaining 26 bits. Does that make sense to people? Okay. Maybe this is the type of instruction that allows any one of 32 registers to be updated; it allows any one of 32 registers to be the base right there, and then it allows all of the other bits to express this as a signed constant. Do you understand what I mean when I say that?

Okay. There are 32 possibilities for this, there are 32 possibilities for that, and there's however many possibilities are allowed based on how much room we have left to encode that, we would need five bits to encode which register gets the update, five bits which determines the base address and the memory offset. And then maybe it's the case that all the remaining bits, in theory, hardware – people who actually still believe would laugh at this, but in theory, the remaining whatever 16 bits could be used to express a signed offset from this right here. Okay. And I draw this subdivision of five and five and 16 right there. That subdivision's only relevant when the first six bits happen to contain all zeros. Does that make sense to people?

For this right here, all I would need to do is set aside five bits. If it read zero, zero, zero, zero, one right there, then it would say, "Oh, you know what? The hardware is implemented in such a way that when there's all zeros followed by a one right here, that the first five bits tell me which of 32 registers I'm assigning to, and all of the remaining bits, all 21 of them, can be expressed a signed integer that actually gets put into that space." Okay. So the subdivision scheme from bit seven forward actually depends and how it's interpreted depends on what the op code says.

Now ultimately what happens at the EE level is that these are all taken as instructions as to how to propagate signals through the hardware so that the signals at the beginning of the clock cycle look like the ones and the 17's and the ten's and the 200's that we've dealt with in the prior examples. Okay. That's the extent of my understanding of EE, the way I just said that. Okay. But you get the principles of what I'm trying to say, right?

Okay. What else did I wanna say? There's absolutely no requirement that all op codes be exactly the same size. You all did – a lot of you did the – I'm sorry, there's this one assignment that we use in 106X called the Huffman encoding assignment. Those who have heard about it know about it. 106B doesn't do Huffman encoding do they? Okay. Well, that's an example where they actually have variable length encodings. Here we have a constant length encoding for all op codes. It could be the case that one of the op codes could just be this right there. Okay. And then some other op codes would have to be longer, so maybe this is another op code. It would only require that the first three bits don't happen to be coincidence if it's something that could be interpreted as a 3-bit op code. Does that make sense to people? That may seem like a silly thing to do, except that this might be the type of instruction that benefits from having lots and lots of bits set aside for some unsigned integer offset. Okay. Or it might be the one that's most popular so it wants the most flexibility in how it expresses its arguments. Okay. I don't wanna say that this is not common; I think it actually is common. It certainly comes up in Nips which is the assembly code instruction that EE's study in EE 108B, I think it is now, and CS majors currently have to as well. Okay. I'm just gonna assume for all of our examples that this is the case, that we have constant op code lengths just because it's simpler to just rely on that type of information. Okay? Does that make sense to everybody? Okay. So there are a couple things I can do in the final six minutes. Before I start talking about function call and return, which I'll get to on Wednesday, I should talk a little bit about structs, but more importantly, I should talk about pointers and casting. That's the part that makes C hard, but somehow compiles. And when it compiles, it means it compiles to something like this, but in – not in CS107 assembly language, but like in X86 or whatever, or, like, Nips or whatever the target language happens to be.

Let me do one example here that's framed in terms of the struck fraction we dealt with lecture three or four. Struct traction int, num, int, denom, and that's it. And I declare a struct fraction called pi, and I do this: pi dot nu is equal to 22. The way I told you that structs were laid out in the third or fourth lecture, that's actually true on virtually – in any architecture that I know of. That the structure packed – that the first field is at the lowest address and everything is stacked on top of it. And if I declare this one variable so that this is logically functioning as my pi variable, then R1, in our world at the moment, stores the base address of the one variable that's there. But the one variable actually knows how it's decomposed into smaller atomic types. Okay. When I do this, not surprisingly, this actually translates to M of R1 is equal to 22. If I do this, pi dot denom is equal to seven, I get M of R1 plus four is equal to seven. So that's easy, except for the fact that you're dealing with structs and you have to understand that the assignments have sort have forgotten about the structs and just are really updating individual integers inside the struct. The part that's interesting, transition to slightly more scary stuff, is if I do this ampersand of pi dot – let me rewrite this. Ampersand of pi dot denom, if I write it that way and then I cast it to be a struct fraction star, I'll abbreviate there, and then I do this. Forget about the assembly code for a second, you know how memory is supposed to be updated; it's a little weird to see this type of thing, but it's irrelevant that it's weird because it's a legal C code, and it's supposed to compile to something. This says, "Identify the l-value of pi dot denom." Where is it located?" Okay. Stop pretending it's a standalone int and think that it's a base address of an entire fraction, go to its ghost denom field and put a 451 there, okay?

The order at which things are kind of realized here, is it discovers this, it evaluates that address, it casts it to think that that right there is the base address of not a standalone int inside a struct, but the base address of an entire struct fraction, and a 451 needs to be placed there. Okay. What assembly code instruction, there's only one of them that's needed, what's the assembly code instruction that would need to be in place in order for that 451 to be placed there? It just translates to this. Now you may think I'm cheating there and I'm actually doing work, but this right here is understood to be an offset of four from R1; the address is four beyond to that base address. Just because I cast it to be a struct fraction star doesn't change the value of the address, it just has a different idea as to what is at that address. It's like the compiler puts on a different set of glasses all of the sudden when it's looking at this one address right here, and it knows that at the denom field that you'd get by de-referencing this, "Oh, it's a struct fracture because it says so," it's an offset of four beyond what it was already an offset from, R1, and then it puts a 451 there. So all the energy that's in place to compile this, either by hand or by code if you wanna write your own compiler, it actually can discover that this is really referring to an integer that's presumably at an offset of eight from R1, which is where pi originally lives. Okay. Does that set well with everybody? So the cast – when you put a cast in place, at least in pure C code, there's no assembly code instruction that gets generated as a result of the cast operation. All the cast does, is it allows the compiler to generate code that it otherwise wouldn't have been able to generate code for because it's like taking a little permission slip to behave differently, okay? Does that make sense? It wants to trust that there really is some legal interpretation associated with the code that it will emit by seeing that struct traction cast right there. Okay?

These are the types of things that exist in the assembly code that's generated by your IMDB get methods and get cast class. The C++ turns out it's not that much more than C in terms of compilation. Okay? You're doing all these void star and car star casts inside assignment three; either you have or you're going to very shortly. The same type of thing happens as it is compiled to either X86 or spark assembly, which is one of the two things you're dealing with if you're on either of the pods or the [inaudible], okay? Does that make sense to people? Okay. I can get arbitrarily complicated with all of these casts. You know I'm the type of person that will be arbitrarily complicated with them, so you'll see some examples of these in a section handout for next Tuesday, and also in the problem set assignment that will go out not this Wednesday, but next Wednesday. It'll be your final assignment before the mid-term, okay? Or I want you to just master this pointer stuff. And believe it or not, you get a lot of mastery by actually coding with it, but if you're forced to draw pictures and generate code and make sure that the code you write down is assembly code, logically matches what mine and the answer key does, it actually resolves any remaining mysteries that might be in place in spite of the fact you get assignment two and assignment three working. Okay? There's still some mysteries that are in place for a lot of people and this usually resolves those mysteries. Okay. Come Wednesday I'll do one more intense example with a little bit more casts; I'll probably bring in an old exam problem and show you how fun they can be. And then we'll move on to starting to understand how function call and return works at the assembly code level, how we introduce the hexagons and the circles to the stack frame and how we get rid of them. Okay. I will see you all on Wednesday.

End of Audio

Duration: 52 minutes

ProgrammingParadigms-Lecture10

Instructor (Jerry Cain):Everyone, welcome. I have a good slew of handouts for you today. Sorry about the lines on the photocopies. It's the just the photocopier, it's not me putting this really annoying background image behind all the text. It's Chapters 3 and 4 of this little computer architecture series of handouts that I'm giving you. I'm gonna spend today and easily the rest of Friday talking about all this stuff and I'll go through plenty of examples.

You also get your fourth assignment today. I'm gonna make it due next Thursday evening. I will tell you now that this is the one that surprises everybody a little bit. It's certainly doable, but there are aspects of using the vector and hash set that always take a few people by surprise, particularly how you store dynamically allocated C strings in these vectors and hash sets. If you've just made one key mistake, then you can actually waste an hour or two trying to figure out why it's not working even though it's compiling.

So be sensitive to the fact that you might just want to read through the handout, and maybe get the first 25 percent of the assignment done because immediately you have to start dealing with these C strings, and once you figure out how to store C strings in these things, you do much, much better and it goes much more smoothly after that. Now I'm gonna do something this week that I won't do very often, but I'm gonna have another discussion section this Friday at 2:15.

Now I'm just inventing the time and I know I can't check with everybody in the class as to whether it's convenient or not, so I just had to schedule it. We're gonna videotape it. We're gonna put it online. The discussion section just happens to come at a kind of crappy time in the assignment cycle. It's like two days before the assignment's due. It's not a disaster for the assignments that we've had so far, but I really want to show you some examples using the vector and the hash set in a structured discussion section, and I want to do it more than two days before the assignment deadline.

So this Friday at 2:15 if you wanna attend live by all means do it. Skilling 191. It's just this week. We will have discussion section next Tuesday as well. I just have this one I wanna insert into the sequence just so you have a little more practice before you really tackle the assignment this weekend, which is when I most of you will start.

Okay. I wanna continue with this cogeneration thing. I wanna get the piece of chalk that I'm gonna use, and I wanna talk a little bit more realistically about what activation records look like. I have kind of blown off the parameters, and where they reside in these activation records. I've only dealt with local variables, but all the functions I've invented have had no parameters passed in. I did that because I just wanted to simplify things.

So there's a couple of confessions I have to make about how I slightly misled you on Monday just to make things easier. If you have this as a function prototype, and you'll be able to infer structure based on this small example, I think. I'll just call it foo, and I'll pass in an int l bar and an int star r called baz, and internally I'll declare some variables. A char array, I'll it snink. I have no idea what these words are; I'm just making them up. And then let's say a short star called Y.

And I don't care about the code at the moment because I just want to show you what the activation record will look like. Now obviously it shouldn't surprise you that this and this and this and this are all packed somewhere close to each other in memory. And in the example I gave you on Monday, I only had stuff like this. The way these are laid out relative to one another does not change. I would have a character array of 4 bytes, this would be this thing called snink, which is a word I've never used before, but here it is.

The four characters would be packed in a static array that resides inside the activation record. Below that, I don't have a short, I have an address to a short that is called Y. Now you may ask what about these things? These things are certainly close. Let me just draw the picture and explain why it looks the way it does. There is this reserved four byte figure that sits right there that has nothing to do with parameters or local variables.

But on top of that there's gonna be bar and on top of that there's gonna be this thing called baz. Okay. And this right here is the full 20 byte activation record that makes up – or that accompanies any call to the foo function right here. Now I put bar below baz. Is that arbitrary? The answer is it is not arbitrary. Now I can't really gracefully explain why this goes below that and why – basically parameters are laid down from high to low address from right to left.

In other words, the zero parameter here is always below all of the other ones. And the first is stacked on top of that and the second is stacked on top of that. Give me 40 minutes of more lecture and I'll be able to explain why that has to be the case in a language like C and C++. These right here are actually stacked in the order they appear. All of these appear at lower addresses than these. This right here is something I'll be able to discuss a little bit more in about 15 minutes.

That is the space that sits in between parameters and local variables. It actually has information about the function that called us. Obviously, foo is invoked from the main function or from some other function or maybe even foo itself if it turns out to be recursive. We're gonna need to lay down a little piece of popcorn right there about where in the code base we actually found this call to foo. Okay. And when the function exits, it relies on this value right there, which I am also going to call a safe PC.

It's what the safe PC value would have been had some function call not interrupted the stream of instructions. Do you understand what I mean when I say that? Okay. But don't worry about how that's manipulated yet. Just understand that it's there and I'll be more sensitive to using it in a few minutes. So that is the activation record layout for an arbitrary function. What I want to do now is talk about how something like that is constructed because a function like foo is called.

Now I think foo is kind of a weird looking function, but I'm gonna go with it. Int main – I'm not gonna concern myself with the parameters – actually I lied. I will. Int arg c char star star arg v and I'm gonna declare one local variable I = 4 and I'm gonna call foo of I and & of I. And then I'll just return zero at the end. Recall back to last Friday when I termed everything in terms of triangles and hexagons. I want to be a little bit more scientific than that.

And I wanna explain how we go from this, where this is arg c and this is arg v and this is set to point to something like 2, and this is set up to point to an array of char stars. Okay. I wanna figure out how we go from that, actually to technically this – I'll call it the safe PC here. When I generate code for this right here, eventually I have to generate code for the function call, you actually generate code to basically allocate space for all the local variables.

Only main's implementation knows how many local variables are needed in order to accomplish what it's trying to accomplish. So by protocol, the very first thing a function does – a C function does, is it makes space for its local variables. Main is called with a partial activation record. It has to complete the full activation record by doing something like this. That's P = SP - 4. Now remember on Monday I was using R1 to always track the base address of the activation record? Okay.

Well, R1 is actually supposed to be a general-purpose register, but there's actually a dedicated register. We just call it SP. It's short for stack pointer. Okay? And that is always the thing that is truly pointing to the lowest address in the stack that's irrelevant to execution. When main gets called SP is pointing right there. The allocation or the creation of this variable right here actually compiles to an instruction to demote this value by four more bytes.

Why four? Because I only have one four-byte figure right here. Okay. And then this becomes the boundary between what's being used in the stack segment and what's not in use. Does that sit well with everybody? Okay. Then it carries on and it does this one initialization right here. The next thing that would happen is that it would just do M. It was R1 on Monday, today it's SP = 4. That takes care of the assignment right there. Okay.

And then I have to actually prepare to call the foo function and wait for it to return before I go ahead and return zero. So the instructions that are executed on behalf of this foo function has to do what somebody did for the main function. It has to set aside space for the parameters. So it has to build a partial activation record for that thing right there. Okay.

It can tell because it sees the prototype how many bytes are gonna be above that safe PC or that return link. Because I have four bytes, and four bytes making up the upper half of the activation record, the first thing that would happen is that you would do something like SP = SP - 8. That would bring this down to there. Okay. This part right there is still the activation record for main, but I've just built 40 percent of the activation record that needs to be in place for foo to run. Does that make sense to people? Okay.

I have to pass in I and I also have to pass in the address of I. I have to make sure that the current value of I is place right there. The address of I is placed right there, and then I have to transfer control over to the code that emulates whatever foo is supposed to be doing. Okay? Does that make sense to people? Yes? No? Okay.

So in R1 I'm gonna do M of SP + 8. In R2, I'm gonna put the actual result of SP + 8. This address right there is relevant to both of those lines. What placed in R2 is the actual value of this thing. What placed in R1 is the contents of this thing right here. Okay. The reason I want to do that is because I want to lay down an M of SP and M of SP + 4 the actual values that need to be communicated to the foo function.

SP is the lower of the parameters. That's the thing that's supposed to get R1. This is supposed to get what I've laid down in R2. Okay. So what happens there is I have effectively done this. I've copied a 4 – actually, a 4 went there in response to that line right there. I copy a 4 right there. I effectively laid down that address in the second of the two boxes. Does that make sense?

Do you understand this part below the arc? It is basically 40 percent of that thing right up there. Yes? No? Okay. After I've done this, what I do in response to after I've set up the parameters, I actually transfer control to the foo function by using this assembly code instruction. That's basically a jump instruction that says jump to the very first assembly code instruction that's associated with the foo function, execute that, and then somehow figure out once you're done executing that to jump back to this right here.

Whatever this is right here, it's gonna actually do this – actually it's gonna do SP = SP + 8. I'll explain why it's that in a second. But do you understand that if I weren't jumping to the foo function that this would be the next thing that gets executed. Does that make sense to people? Okay. This address is actually, what gets laid down in this little safe PC thing. Okay. And it's automatically placed there by the call instruction.

At the time that the call instruction executes, it has a clear idea of what PC is so it knows what PC + 4 is. It actually on our behalf decrements SP by four more bytes and lays down that safe PC right there. So when foo is done executing it has information in its activation record about where to jump back to. Okay. This is basically at the electronic level or the hardware level, basically a piece of popcorn to remember where you were walking before you took a turn. Okay.

Does that make sense to everybody? Okay. This transfers control over to the foo function. Well all I'm gonna do as part of foo, is I'm gonna do something like let's say Y = short star snink + 2 and then I will do *Y = 50. And then I will return. Okay. What foo has to do is it has to complete its activation record by decrementing the stack pointer even further to accommodate and make space for those two variables right there.

So what happens is that this – let me actually write this somewhere else. What foo does, it makes space for these right here and that right there. What happens is SP is set equal to SP – 8. Why is it minus 8? Because it can tell while it's being compiled that that's how many bytes are needed to extend the activation record distance enough just to complete the full activation record. So it does this, brings the stack pointer down to there, leaves it unitialized. Okay. You know that because C isn't an initialized local variables so the assembly code won't either.

And then it carries on and compiles code for this, and it accesses this variable and that variable relative to the value that it's the SP register. So the first thing that happens here is the value of Y, the contents of this thing right here, has to be updated to include the address of that thing right there. Okay. Does that make sense to people. Forget about the cast, I have to evaluate what snink + 2 is. That's really & of snink + 2. Okay.

So what I can do, in R1, I can set that equal to SP + 6. Why that? SP points to the base of the full activation record at this point, has to go four bytes beyond to basically circumvent Y, and go two bytes forward because it knows that snink is a char star so pointer [inaudible] the same thing, and it wants to store that address at this moment into a register so that it can be assigned to M of SP.

So R1 stores that value, M of SP identifies this space right there. That's what we want to get this value because that's the space that overlays the variable called Y. Okay. Make sense? As far as this is concerned, what has to happen is I have to get a 50 somewhere in memory. Where in memory? Well it's at whatever address happens to be stored in the Y variable.

So what would happen here – that separates the variable declaration. There's that. Into a local register, I would actually reload Y, R1 stores the address where a 50 should be written, but I only want to write it as a 2-byte figure because it's typed to point to a 2-byte figure. Does that make sense? Okay. So I would do M of R1 = .2, 50. Okay. Does that make sense to people? This is done. Okay. So now, it has to exit and basically return somehow to the main function that's right here. Okay.

You understand why that SP = SP - 8? Sits right there. That was to make space for the two local variables to basically make use of eight more bytes in the local stack segment for its local variable set. Well it has to promote SP back to where it was before it entered this function. This is basically the equivalent of – I don't want to say it's the equivalent of malloc; it's the equivalent of allocating space for variables, we have to deallocate that space. Okay.

So for SP=SP + 8 would be the last thing that's done right here. That leaves SP to be pointing right there. Do you understand that SP is addressing the very 4-byte figure that has the little piece of popcorn in it? Okay. Does that make sense to people? So this final instruction, RET for return, is understood to be an assembly code instruction that pulls this value out, places it – basically populates the true PC register with what this is, brings the SP register up four bytes and then carries on as if it were executing at PC + 4 all along. Does that sit well with everybody? Okay.

So there's that. As far as main is concerned, this is the next instruction that gets executed after this return executes. Okay. It jumps back, and it puts the address of that in the PC register. This SP is equal to SP + 8. Actually deallocates the space that's set aside for these two parameters that it put down there in preparation for the foo call. Does that make sense? Okay. So that's what that is, and then what I do, is I use another register. There's not too many dedicated registers with special names, but this is one more of them.

We have PC, we have SP, we also have this thing called RV which is this four byte register dedicated to communicating return values between caller and callee functions. Okay. Does that make sense to people? Okay. I'm returning a zero to whoever called main, so you have to think of RV as this little cubbyhole where return information is placed so that once it jumps back to whatever function called main, it knows to look in RV immediately, and pull that value out to take it as a return value.

The metaphor I usually use here is that think about you're trying to leave money in a locker at an airport. Okay. You wanna put the money in the locker, and immediately walk away when you know the person who's supposed to get the money is the next person looking at it. Does that make sense? Okay. We didn't have a return value for this foo function. I'll write another function that's a little bit simpler than this that really just makes use of a meaningful return value. This is also a meaningful return value, we usually just blow it off because we're not concerned about who's calling main.

Let me just draw a general activation record. Here are all the params. Here's the safe PC where the return link, and here are all the locals. These are allocated and admitted by the person calling the function. These are allocated and initialized by the actual function that's being called, the callee. So the entire picture is the activation record that has to be built in order for the code inside a function to execute and have access to all the variables.

It may seem a little weird that this part, and technically everything through that right there, but that it's set up and initialized by the person calling the function, and the rest of it is set up and initialized by the function itself. Why is there this separation of responsibility between actually building the entire thing? Why can't the caller build the entire thing? Do you understand what I mean when I say that? Why couldn't main set aside space for all the variables?

Why couldn't foo actually set aside space for all the variables? The reason is that the caller has to be involved in setting at least this portion up because only it knows how to actually put meaningful parameter values in there. Does that make sense? Who else is going to put the four and the ampersand of I in there other than the caller. Make sense to people? This right here can't be set up by the caller because the caller has no idea how many local variables are involved in the implementation of, in this case, foo. Okay. Does that make sense?

So the separation of responsibility really is the most practical and sensible thing to do. The caller knows exactly how many parameters there are, can look at the prototype to tell, and it knows how to initialize it. That's why the top half is set up by the caller. The bottom half, the caller doesn't even know how many local variables there are much less how to manipulate them.

So you have to rely on the callee function, this is foo in this case, to go ahead and complete the picture by deallocating SP to make space for the local variables. Now I put this right here. When I say call foo right there, it's really a jump instruction that's associated with the address of this first instruction right here. RET is an instruction to basically jump back to whatever address happens to occupy the saved link. Okay. Sit well with everybody? Okay, very good.

Let me write a little bit more of a practical function just so you understand how this return register and call and return all work in a case of a recursive function. Okay. So let's write a function that's a little more familiar to us. I want to write this function called factorial which is framed in terms of a single parameter called N, and I'm not gonna have any local variables. If it's the case that N double equals zero, I just want to go ahead and return 1.

Otherwise, I want to go ahead and return m times whatever the recursive call to factorial of M – 1 returns. I'm not concerning myself with M being negative. I don't care about the error checking. I want to concern myself with how this translates to assembly code in our little language. Okay. Let me erase this. Yep?

Student: Okay, in the first [inaudible] of foo, you substituted [inaudible], I mean are you insinuating that you can substitute [inaudible]?

Instructor (Jerry Cain):In other words, do one variable at a time?

Student: Yes.

Instructor (Jerry Cain):Yeah, you could do that. It wouldn't be incorrect. Compilers when they're generating this code, they can look at the full set of local variables that are declared at the top, and it can compute the sum of all the sizes, and reduce the allocation of all of them to one assembly code instruction instead of several.

Student: So it's more efficient, right?

Instructor (Jerry Cain):Well, yes, technically. I don't want to say that it's – that's not a top priority, but real compilers would just use that right there because they could very quickly add up all the variable sizes, and just know that they're gonna be packed together, and just do this on one little swoop. Okay. Now as far as this factorial function is concerned, I want this to label the first assembly code instruction that has anything to do with this as a function.

It has to assume this as a picture. It has to assume that some value of N has already been passed in. Okay. That whoever called it, laid down a call to the fact function, and as a result of that, laid down some safe PC so that it knows where to jump back to after factorial computes its answer. Does that sit well with everybody? We can momentarily forget about the fact that function, call, and return is involved because the first few statements right here are just normal little C code. Okay.

What I want to do up front is I want to load the value of N into a register, SP + 4 is the address of M in this case. Okay. And I want to pull the 4 into a register called R1 because I want to conditionally branch around this return statement if some test passes. Okay. Well, I want to branch – I'll leave those open for the moment, depending on whether R1 and zero basically mismatch. If it's equal to zero, I just want to fall to the return statement and scram, but if they're not equal, which is why I will put M and E right there.

If they're not equal, then I'm doing a little transition of this, then I want this to not execute and it to come down here. So all I'll do is I'll put PC +, and I'll leave this open because I don't know how many assembly code instructions I'm jumping forward yet until I execute the code for that. Okay. If this test fails, it's because this test passes and I want to basically populate RV with a 1 and then call return. So I would do this return value is equal to 1 and then I would return. Okay. Does that make sense to people?

If this doesn't happen – I'm sorry, if this test fails, I better jump beyond the return statement, which means that this should be a 12. Okay. And the next instruction I'm gonna draw, has to start on the computation of this thing right here. Now I have no reason to load M into a register because I'm only going to potentially clobber that register when I call this function recursively.

So what I need to do is I have to prepare the value of M – 1, push that onto the stack frame in preparation for a recursive call to factorial, let the recursive call do what it needs to do, assume it puts an answer in the RV cubbyhole, and then use that answer immediately to multiply it by what my local value of N is to figure out what RV should not be populated with. So I will do this, R1 = M of SP + 4, that loads in into R1. I will compute what

N 1 is.

And now I have all the information I need to make this recursive call. So I'm going to set up the partial activation record. I'm gonna do SP = SP - 4. Okay. I'm gonna write to SP, this value of R1. So what just happened here? I decremented – this is where SP is pointing right now, I decremented to a point right there. That right there is the original activation record. I'm in the process of building the activation record for the next call. I lay down a 3 right there, because that's what R1 has at the moment. It had a 4, but I just pulled it down to a 3. Okay. Make sense?

And then I call on the factorial function. In response to that, this is decremented four more bytes, the address of this instruction is placed in the safe PC, but execution jumps back to follow this recipe all over again, but on behalf of a second activation record. Does that sit well with everybody? Yes? No? Okay.

So it's as if the primary call here – you just can think of it as suspending, it's not really what's happening, but access to this activation record is temporarily suspended while the recursive call deals with these addresses downward. You just have to assume by protocol, that the recursive call is the person you're in cooperation with and the recursive call is placing money in the RV locker, okay, so that when you pick up execution right here, you know that the RV register has something meaningful. Okay.

Well, when you get right here, you have to clean up space for the local parameters. The return statement that brings us back here, gets rid of this, and then that SP is equal to SP + 4, brings this arrow back up here, and that was the picture that was in place before we involved any type of recursive function call or any function call at all. Okay.

We know that RV has – it would be six if factorial was really doing its job. So I'll actually put that. It has a six in it, if we just take the leap of faith, but at the assembly code level that the recursion is working, and then once I clean these up right here, I have to reload M, so R1 now has a 4 in it. I can do this. That emulates that multiplication right there. Okay.

RV has the return value of this. R1 has the value of that. I have no local parameters to clean up. RV has the register that I need to communicate back to whoever called me, so I will just return. Okay. Does that make sense to people? Question right there?

Student:[Inaudible]

Instructor (Jerry Cain):The PC + 12, where did I put that? I'm drawing a blank. Oh, this right here? This is normally, if there are no branch instructions, you know how PC is the address of the currently executing instruction, and each instruction is 4 bytes wide. So by default with each clock cycle, the PC value was updated to be 4 bytes larger than it was before. So normally it executes things in order.

If this is PC, this is PC + 4, PC + 8, PC + 12. If this branch instruction actually passes then I don't want to fall to this, you specify as the third argument the actual address of the instruction you should jump to relative to the current PC value. Okay. Now in some real systems, PC's already been promoted by four, so this could be PC + 12 on some other systems, but in our little world I'm just assuming that PC retains its value during the full execution of this thing right here. Okay. And it's just identifying that right there. Were there other questions? Yeah?

Student: What it be alright to do something less arbitrary –

Instructor (Jerry Cain):Yeah, then it's a little different. I'm gonna just constrain all our examples that deal with things that are exactly 4 bytes. I'm sorry, 4, 2 or 1 bytes; things that can fit into an RV register. MIPS as an architecture, I know that one best of all of them, there are two return value registers and it usually only uses one of them unless you're returning like a double or a long, long, which is an 8 byte integer, and it would use both of them.

If you're returning a struct with 12 or more bytes in it, then what it'll do, is it'll actually place the address of a temporary struct somewhere in memory, and just assume that the caller knows that a struct is being returned and will take the RV register and dereference it to go to the thing that's actually a struct. Does that make sense? So they figured it out. I'm just not gonna deal with that particular stuff because it's more minutia to worry about. Yeah?

Student:[Inaudible] I didn't quite get that.

Instructor (Jerry Cain):That's emulating that multiplication right there. At the C level you know you have to spin on the return value from the cursive call, and multiply it by the current local value of M. Right?

Student: Right.

Instructor (Jerry Cain):So this right here, just stores the current value of M and RV right there, stores the result of the recursive call. Okay. Does that make sense? Now I have the first piece of evidence as to why I always want you to reload variables. Do you understand that right there I absolutely had, absolutely had to reload the value of M. Okay. You may say, well I did it right there. Okay.

And even if I didn't do that right there, I did this with R2, I'd absolutely have to reload the value of M because I have no idea how complex and how motivated this as a recursive function, or any function at all, is to actually clobber all of the register values. All of the function calls are using the same exact register set. Does that make sense? Okay. So that's why I want you to get in the habit of reloading all your variables as you advance from one C statement to the next one. Okay. Does this make sense to people? Yeah. Question right there? Student:

Why do [inaudible] right after the return statement above?

Instructor (Jerry Cain):Because – this right here, for the same reason basically. I want all code emission to be context insensitive and I don't want it to leverage off of things that happened in prior C statements. Real compilers would do that. I just don't want us to do it. I'm not adamant about it, but I just think that there's a nice clean formula about always generating things from scratch because if this code changes because the C code statement prior it changed, this code still does the right thing for the statement that it's being admitted for. Does that make sense? Okay.

So I don't want to kill this example yet. I want to run through an animation, which is why I have my computer here just so you see how the stack grows and shrinks in response to assembly code instructions actually running. Yep?

Student:[Inaudible] why, for example, isn't there a PC - 12? Or not 12, PC – [inaudible]

Instructor (Jerry Cain):Oh, I see what you're saying. When the program is actually loaded, this actually would be replaced by PC - 28 or whatever it is. These are just placeholders at the moment because these are easier to deal with. Think of these as like go-to labels. That's really what they function as at the assembly code level. A real assembler or linker would in fact go through all the dot O files, and replace these things right here with the actually PC-relevant addresses. It usually will defer it until link time because it wants to be able to do the same replacement of these things with PC-relevant addresses between functions that are actually in different dot O files. Okay.

And I'll talk a little bit more about that on Monday, and probably Wednesday about how all the dot O's are basically assembled into an A dot out file or a 6 dash degrees or what have you. Okay. Okay. So give me two seconds to do this. Okay. We should be good to go. Create some mood lighting. Let's see how well this translates to the screen. Okay. So this is more or less – it's an exact replica except for spacing of the function I just wrote on the board. Okay. And, oh, that's not showing up. That's not good.

Can I get it to show up? It shows up over there. That's great. Oh, it's really up there. That stinks. That's never happened before. Hold on a second. Maybe it'll just readjust to the screen size. Stalling, stalling, stalling, stalling. It has no input so something's weird going on. Okay. Actually, I have an idea. Yeah, but that's still not really very good. Let me do – let me just run it in place. Okay, you guys can all see this right here. Let's make sure the full picture can be seen. What's up?

Okay, there's that. It's actually pretty close. I don't think it zooms very well when it's driving both this computer and that monitor. Okay. That's actually not bad. The return type of fact is int. Okay. It's being clipped up by that mushroom that's on the screen up there. So this right here is the code – the C code obviously. This right here is the assembly code I more or less just wrote on the board. Okay. This is the animation, and I'll try and do this this way. Let me do next slide. Yeah, this'll work. There's that. Okay.

So the original call is the factorial of 3 and it's just gonna follow these instructions one by one and use the accumulation of the stack to just compute what 3 factorial is supposed to be. Now it checks immediately. It loads in. It checks to see whether or not N = 0. It's not, so it actually does branch forward and it prepares for the recursive call. Does this make sense to people? Okay. It has to assemble the recursive value, has to make space on the stack frame for that new – the recursive value of M, it has to write to there.

So you how we're basically in the process of building the activation record for the recursive call and then we have to call factorial and some things are updated. The little circle that's been populated in the safe PC portion, it actually has as a piece of popcorn, a pointer to the next instruction so it knows where to carry forward when the recursive call returns. Does that make sense? Okay. Question?

Student: So we aren't required to store PC?

Instructor (Jerry Cain):No, it actually happens – the call instruction does that for us. Okay. Do this – comes back up here – the original call doesn't forget where it should carry forward when the recursive call returns because we have that piece of popcorn in the stack frame. Okay. So there is this is exactly the same thing where N = 2 and N = 3. Notice that both recursive calls have as their safe PC the instruction after the recursive call. That should make sense. They all need to know where to carry forward from after the call returns.

It just happens to be the same place both times. It does go through one more recursive call, lays down a zero, one more, and then finally it comes to an invocation. It's following the recipe for a fourth time. The first three times it's suspended execution, but this time this branch instruction is going to fail, so it's going to go and actually populate RV – actually I can't – this right here has been updated with a 1, and so it's gonna go ahead and return these second to last recursive call is like, oh, I'm active again, I'm going to continue carrying on from where I was left off before.

I immediately look into the RV register and spin RV 1 x 1 stays the one, but as things unwind, the all carry off from here, the 1 in RV becomes a 2 – this isn't working very well – the RV becomes a 6, and finally I return to whoever made the original call which is probably – or which may not be a factorial. Does that make sense to people? Yep?

Student: Those arrows that were pointing down onto the screen, where those –

Instructor (Jerry Cain):They were the safe PC registers. The safe PC blocks. You remember the 4 bytes that existed between parameters and locals? Well there is a real number that's placed there. It happens to be the address of the instruction that comes after the call statement so it knows where to jump back to. It's almost as if it pretends that it wishes the call thing were just one assembly code instruction, but since it isn't and it jumps away only to come back later on, it needs to know how to come back. Okay. Okay. So does that sit well with everybody?

So that was a breeze through that, but I have a little bit of stuff to talk about come Friday. The one thing I will tell you is that C++, if you code in pure C++, you're probably programming in the object-oriented paradigm so your object and data focused as opposed to function and verb focused. In C, which we're dealing with right now, you always think about the function names, the data is incidental, you always pass the data in its parameters as opposed to having the data being messaged by method calls.

We're gonna see on Friday, and this will be an aside, I won't actually test you on this part on the mid-term, but you'll see that C++ method calls and C function calls really translate to the same type of assembly code that everything is emulated by a function call and return with call and return thing at the assembly code level, and it's really just an adaptation of either object orientation or imperative procedural orientation to assembly code to get it to run, emulate either C code or C++ code. Okay. And that's what we'll focus on on Friday. Okay. Have a good night and I will see you on Friday.

[End of Audio]

Duration: 47 minutes

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

ProgrammingParadigms-Lecture12

Instructor (Jerry Cain):Here we go. Hey, everyone, welcome. I actually have two handouts for you today. They're posted online, and we'll distribute them through the lecture at just right now, as I'm starting. One of them is tomorrow's section handout. Its focus is pretty much on the assembly code generation I was talking about last Monday, Wednesday, and Friday.

The second of the two handouts is Assignment 5. I was gonna hand it out on Wednesday. Then I said, "You know what? I'll hand it out a little bit earlier," because I just want to like afford everyone the flexibility to work on these problems when they have time.

You don't have to hand anything in for Assignment 5. It is a written problem set. There are no programming exercises whatsoever. It's just lots and lots of practice with this code generation stuff that we're doing in section tomorrow, but you'll also certainly see C code generation on the midterm next Wednesday evening.

So the only deadline I'm really imposing on Assignment 5 is that you actually do the problems, make sure your answers are consistent with mine. And I say it that way because it doesn't have to be exact on an instruction by instruction basis, but you have to just make sure that your code dereferences things the right number of times and loads things the right number of times in order to feel comfortable with that material because it definitely will appear on the mid-term I give next Wednesday evening at 7:00 p.m., okay.

I don't know where the mid-term is gonna be yet. I'll probably figure that out in the next two days. This Wednesday I will certainly give out a practice mid-term and a practice solution so that you have some fodder to play with over the course of the next week. But all of the section handout problems, if there were 20 problems on those section handouts, 19 of them came from old practice midterms.

So definitely make sure you understand those. You're welcomed to bring in any lecture notes, any of your assignment printouts, whatever you want to bring in. You can bring in textbooks. I don't see the value of it since everything that you're really responsible for has been covered either in lecture or in handouts. So that's that.

I know Assignment 4 is due this Thursday. I think people have started it, and they are true believers when I say that it is probably the most difficult of the four you've seen all quarter. So start that soon, even if for no other reason than just doing a small component of it tonight so you know what you're up against with this Thursday deadline, okay.

When I left you last time, I had just started to talking about the C preprocessor. I want to talk about preprocessing versus compilation versus linking. You're used to, from at least 106 memories, it all being the same thing. You clicked command all or you did a drop down and you clicked build, and all of a sudden, this double clickable app was created.

That's because it does these three things in sequence behind the scenes, and it doesn't very clearly advertise whether or not something in preprocessing or compilation or linking broke down. You don't necessarily know the difference.

So I want to focus on the differences and tell you what each phase is responsible for. And when I left you last time, I had just introduced the notion of a pound define, and I advertised it quite clearly as something that was no more sophisticated than glorified search and replace of this token with that text right there. So if I do this, a height – whoops, and I say that this is 80, then anywhere K width and K height appear beyond these two lines, it actually substitutes this for that and this for that.

The only exception is it won't do that in string constants, but it'll even do it in future pound define. So if I were to do something like this, K perimeter, and I equated it with K width plus K height, then this would not only substitute anything down here, but it really would replace that with a 40, and this right there would be replaced with a 80. So by the time you got around to the definition of K perimeter, it would see this not as this token stream, but as two times open paren, 40 + 80, close paren, okay.

It doesn't evaluate it. It doesn't even recognize that they're integers. It just looks at it as blank text, but the substitution of this there and this there is exactly what you do want. Pound defines are really nothing more than glorious search and replace. We use them in C, pure C, to consolidate what otherwise metric numbers and metric string constants, and attached meaningful names to them. What you may not know about pound defines is that you can define an extension to the pound define, and you can actually pass arguments to pound defines as if they're functions.

They're not called functions. They're called macros, so I could do something like this. The maximum string, A, B. As long as there's no space between that paren and the final character of the token right there, it's clearly understood to be a little bit more than just a pound define constant. It's a pound define expression that's parameterized on whatever A and B adopt in context when they're used later on, okay.

So if you want some quick and dirty way to find a larger of two numbers, you could substitute it with this. And just to be clear about order of operations and evaluation of everything, we usually see an intense number of parenthesis put around these things just so that there's absolutely no ambiguity as to how things should be evaluated if this thing is just plopped in context somewhere later on, okay. And order of operations might otherwise confuse things. You'll actually see an example of that in a second.

Anywhere you see this later on, you wouldn't type it in this way, but just pretend that you did. If I, for whatever reason, needed it to tell me that 40 was, in fact, greater than 10, when I see this in code later on, it really will go and find the max symbol, and it will – every place that there was an A, it will place a 10.

Every place there was a B, it will place a 40. So this would, during preprocessing, be replaced by 10 greater than 40, if true, 10, otherwise 40. And even though that's an obtuse way of identifying that 40 is greater than 10, that is the textual substitution you would get in response to that, okay.

So it's like a pound define. It's this quick and dirty way to inline functionality that's otherwise complicated with something that's a little bit more readable. You could, of course, go with a function, but you already know from the assembly code you saw last week regarding function call and return, that a lot of time is spent setting up parameters, writing the parameters there, jumping to the function, and then after it's all over, jumping back and cleaning up the parameters.

It's not that much work. It may be ten assembly code instructions, but this is the type of thing that would expand to like three or four assembly code instructions. So the entire function or the entire effort of determining a maximum number using just traditional function column return would spend 70 percent of its time, or something about that percentage, just calling and returning from the function. Do you understand what I mean when I say that?

Okay, using this pound define thing, this is this very efficient way of jamming in an expansion of this every place MAX with two parameters is actually used. Now it doesn't actually require that A and B be integers. I mean, of course, we know to look at them, that they should be integers, but if I were to do this, get rid of that.

If I were to be senseless and do something like this, this would eventually cause problems. But as far as preprocessing is concerned, all it would do would be doing – all it would do here is do templatized search and replace, would use this.

Every place there's an A there, you'd see a 40.2, and every place there's a B, you'd see a hello as a string constant. And only during compilation when it reads the expansion of this as if we typed it in that way, well, say, you know what? I don't like you comparing doubles to car stars, okay, using a greater than sign. Okay, so you would get in there eventually, but you wouldn't get it via the preprocessor. Do you understand what I mean when I say that? Yep.

Student: Is there a good or bad style to do something like that?

Instructor (Jerry Cain):Well, in something like this?

Student: Yeah.

Instructor (Jerry Cain):I actually don't see the problem with this as long as you have been doing it for more than a few days. I mean this is – I'll show you an example of two pound define macros that we used in Assignment 3, one of which you didn't even know you were using, and the other one is in my solution, okay.

This is obviously a hack just to introduce a point, that preprocessing is still just text and replace, and that it leads to problems later on might be tracked and flagged in compilation, or it might be flagged when you get a (inaudible) at 4:00 in the morning. Okay, you just don't know. There was a question right here.

Student: Do you receive from this (inaudible)?

Instructor (Jerry Cain):The question is do you receive anything. It doesn't receive in the sense of return value, but this is an expression that evaluates to either the result of evaluating A or evaluating B. So this one, before I crossed it out, this would evaluate to the number 40. So if I wanted to, I could do this, all caps, of like, let's say Fibonacci of 100 and factorial of 4,000, and I'm curious as to which one's bigger.

There's actually a problem with that that I'll outline in a second, but that would really bind max to the larger of those two values, okay. It's interesting that this is something – there's something about that call that I don't like, but I'll explain that in a second. Let me just show you some reasonable uses of pound defines. We'll be more central.

Do you know how in Assignment 3 there was some situations where you wanted the assert condition to be either greater than or equal to zero, and less than logical length, and in other ones, you wanted to be less than or equal to logical length? And depending on how aggressively you reuse and call vector nth yourself, there may have been situations where you were blocked out by the assert statement that sat at the top of the implementation of vector nth.

Vector append or vector insert, the logical length is a completely reasonable parameter to accept, but if you called vector nth using that value and you had the right assert statement inside, it would actually block you out and error out and end the program. Do you understand what I mean when I say that?

Well, what I did, rather than writing a function that computed the nth of the address of the nth element in a blob of memory, I wrote it as a pound define macro. I just did this. Pound define, I called in nth Lin address, and I framed it in terms of base and a Lin size and index, okay. And I equated that all in the same line. You can actually do that and it'll allow you to continue the definition on the next line. I equated it with this like that, okay.

I could have written it as a function. The reason I wrote it as a separate thing altogether is because I wanted something that did the point arithmetic for me without the asserts. I wanted to control the go and get the millionth element, even if it were dealing with an array length too, but I would actually call this from within vector nth after I've den the assert. Does that make sense to people?

And so this way I had this quick and dirty way of actually doing this type of point of reference just once, studying it and saying, "Okay. This needs to be careful code because it's the type of code that can go wrong if you're not careful about it." Make sure that this is doing exactly what I want it to do, and then call this everywhere. I see a lot of people do the point arithmetic like seven or eight times in vector dot C, okay. And if you're cutting and pasting it, that's not great, okay.

If you're cutting and pasting and you got it right the first time, it's probably okay, but I'd much rather see people consolidated this to either a help or function, or now that we know it, a little macro that jams this calculation in the code for me even though it looks like the function, okay. Does that make sense?

There's no asserting going on here whatsoever, so I can get the asserts right, and rather than calling vector nth everywhere, I can just call nth a Lin address, okay, wherever I would otherwise call vector nth internally. So I never have to worry about whether or not the off by one nature of what vector nth allows in terms of incoming values to block me out accidentally, okay. Make sense?

Now the thing about this is this looks like a function call. There is really no type checking done on these things right here, so this only works post preprocessor time. If this gets specified to be a pointer, and these are things that can be multiplied together and ultimately be treated as an integer offset, okay.

You usually do get that right, but it's not as good as a true function in that regard because a preprocessor doesn't do error checking at all, but it does push the expansion to the compilation phase where it does do error checking, okay. Usually don't like separation of cogeneration, or I'm sorry, let's say C code generation from the actual type checking, but you just deal with it with the down points. Question over there?

Student: Well, with the (inaudible) equal to or as integers?

Instructor (Jerry Cain):You could certainly. When I used this, I implemented void star vector nth, took a vector star V, and an nth. I think it was called position. And as it turns out, it was two lines long. I had the assert position greater than or equal to zero. I had the assert – I actually had these on one line, but I'm just making it clear, position is less than V arrow lodge length, spell it right, lodge length. And then right at the end, I said return nth a Lin address where I passed N V arrow OM's, V arrow OM size and position.

So in response to this macro call right there, it's not really a call. That's kind of the wrong word. It's just the placement of a macro so that it expands during preprocessor time to that as if we typed it in ourselves that way, okay. And as an expression, it evaluates presumably to the right address, so that's what gets returned, okay. That make sense? Okay.

There are some drawbacks to this. It's quite clear that that's a macro because I put it right there. What you may not have know is that these right there are also macros, okay, and I'll show you what they look like in a second. They're a little weirder, but nonetheless, they are in fact macros, and that's how they can be stripped out using some compilation flags so that they're not present in the final executable that you ship as a product. Somebody had a question, yeah.

Student: Quickly, that's not your data void star, but the top was car star (inaudible)?

Instructor (Jerry Cain):Actually, in pure C it's not a problem. Actually, in none of the language it's a problem because remember void star is like the all accepting pointer, so it's what you're doing when you assign something of type car star, which is what this becomes, and you return and funnel it through a void star. You're doing what's called upcasting. You're just going from a more specifically typed pointer to something more generic, and it just knows that there's no danger in that direction.

It's when you downcast and you say, "I have this generic pointer, but now I'm claiming that it was really this very specific pointer all along," but you really do need a cast in many situations, certainly, if you have references involved, okay. Other questions? Okay.

So the problem with this that I did outline is that you don't get side checking at all during the preprocessing phase. There are other problems associated with this, but let me talk about what assert looks like. You've seen – I imagine 80 to 90 percent of you have actually seen an assert statement fail, and you've seen what happens when the condition is passed to assert isn't met.

The assert dot H file defines assert. It doesn't define it as a function. It looks like a function call, but it really is this. Define assert, and I'll just put C, O, N, D, and it's equated with this. It actually evaluates cond, okay. And if the condition is true, you know that it just returns in the functional sense, although it really is not a function call.

When this passes, it just basically evaluates to a no op and doesn't do anything, and just continues to the line after the assert. What it does is it needs to have at least one statement in the – sort of the if region of this turnery thing. So it just casts zero to be a void just to say, "Okay, don't do anything with this zero. Don't allow it to be assigned. Just has to be present to sit in between the question mark and the colon."

This right here is some elaborate thing. That printout, it's a standard error. Some string that involves the filename and the line number of this assert in the original file followed by an exit. Actually, it doesn't have this right there, okay.

So you may not understand the syntax and how everything is exactly relevant to the implementation of assert, but you know that this looks harmless and this looks pretty drastic, okay. So whenever you put assert position is greater than zero in your code, what you're really asking the preprocessor to do for you is say, "Yeah, take this assert position greater than zero, greater than or equal to zero, and replace it with position greater than or equal to zero, oh, awesome. Don't do anything, otherwise end my program and tell me what line this thing failed at."

Does that make sense? Okay. The actual full definition is this. If defines N debug, that's kind of like a pound define, but it's an if question about the presence of a pound define. If that's the case, then pound define assert of condition to just be a no op – whoops, regardless L rather, we'll do this. So this is the thing you're using in Assignment 3, and this is the way it's – this is really turned on.

If you pass or you define a count defined constant prior to this called MD bug for no debugging, then it replaces all of your assert calls with this harmless statement right there. Okay, so it technically is one statement, and this zero compiles to just one line of assembly that's optimized down to zero lines of assembly. But that's how the asserts go away when you compile it a different way so that there's no danger of asserts actually failing on your behalf in production code. Does that make sense to people? Okay.

There are some problems with the definition of assert, not really. I actually want to go back and revisit this function right here, and in particular, that right there, that particular use of max, and start to show you the drawbacks of the preprocessor. And this is actually related to why I prefer static const globals as opposed to pound define constants because I'm trying to like get you away from the preprocessor to the degree you can.

This right here, it is so literal about a textual search and replace that it will call one of these things once, and the other one, the larger of the two might quite arguably be the more time consuming one. It will call it twice. Why is that the case?

Because this right here, because of that pound define definition for max over there, that one expands to this is equal to – this is the case that Fibonacci of 100 is greater than factorial of 4,000. If so, then return Fibonacci of 100, else return factorial of 4,000. Okay, and then there would be that right there.

Okay, that's how literal the text and replace is, text, search, and replace is. And so you actually get the imprint of this is a very time consuming function. This isn't quite as bad because it's a linear recursion. But if you turn it up to Fibonacci 100 is greater than this right here, it's gonna take not only a long time, but twice as long as a long time, okay, because of the second call right here. Make sense?

So it doesn't actually cache results internally, or it's not clever at all. It assumes that you really meant to type it this way because of the way you framed the definition of the pound define, okay. There are even – even so, even if it's kind of stupid from an efficiency standpoint, at least it's correct. Clever C programmers at one point go through this phase where they try to do as much in a single statement as possible, and so they might want to figure out the larger of two variables, and simultaneously increment the two variables, so they'll do something like this.

Oh, yeah. I want to know the larger between M plus M and N, but I also want to increment both of them at the same time, okay. It will actually commit to a ++ on the smaller one just once, and will commit to a ++ on the larger one twice because of the way it expands it. This would be replaced at preprocessor time with this. Is M greater than M? Oh, and by the way, increment them.

Whoops, oh, it is, okay. Well, then return the value and then increment it, otherwise, return the other element and increment it. So you certainly see that ++ is being levied a total of three times, okay. That make sense? It'll return one more than the true larger value, and it'll also promote the larger value twice as opposed to once, okay.

Now you could argue that these are moronic examples because people wouldn't do this in practice, but you could also argue that the language should be sophisticated enough that it just doesn't allow people to do these types of things because if it does happen, maybe it happens one day out of 300, once a year, but you could very easily spend four to eight hours just trying to figure out why this one little line isn't working properly, okay.

When those types of things are allowed to happen, you have to somewhat blame the language. You certainly can blame the language as opposed to the programmer if other languages wouldn't have allowed something like this to happen, okay. So as we get to be better programmers, we'll start to be more opinionated about how good the languages themselves are, and how they allow us to quickly get to a final product, and making it as easy in the process as possible.

Okay, C is really working against you in a lot of ways, okay. It was invented in like the late 60s, early 70s. It came into fashion. The spirit of programming then was let me do whatever I want, man, and so you can get down in the hardware. And it wasn't as problematic then because think about how small code bases were in 1965. You can't even think about that because I wasn't even born yet, much less you.

But you're dealing with programs, except for operating systems. Unix was being written in the late 60s and early 70s, maybe a little bit earlier than that, but most programs were like pawn and maybe like miniature golf with like the most ridiculous paddle – club and ball that you can imagine, just really, really simple programs that had to fit in 64K of memory or 16K of memory. There just couldn't be that many programs. That just means programs were more manageable then.

Now you're dealing with code bases. I can't even imagine how many lines of code exist behind Google walls, behind Microsoft walls. We're talking millions, tens of millions of probably lines of code, probably more than that. I have no idea, okay, but like we're a magnitude like where the exponent is six or seven, okay, very, very large. If you're weighing that much code, you don't want to have to say, "God, this," you don't want to have to look for a problem like that and do a binary search on 10 million files to figure out what the problem is.

You want it to be very, very likely that you'd get something right the very first time you type it, and that is unlikely in C++. You're all learning that right now, okay. Does this make sense to people? Okay.

There are other aspects of the preprocessor I should talk about. I think I've hit on everything with regard to pound define. There's also the pound include. When you do this, pound include, I'll do assert dot H. I'll do one above it, include – let's do STDIO dot H. That's for print F and scan F and things like that. You know about assert dot H, and then you also do this, and you saw things like how to include gen lib and simp IO dot H in CS106. I don't know whether anyone ever answered the angle bracket versus the double quotes thing, whether you just say, "Oh, I have no idea, but I'll just do it because it works if I do it that way."

Whenever you use angle brackets or less than and greater than signs to delineate the name of the dot H file, it's taken by the preprocessors to mean, oh, that's a system header file, so that actually should be with the compiler, so I should look one place by default for those files. But when it's in double quotes, it assumes that it is a client written dot H file, so it looks in the actual working directory by default.

There are flags you can pass to GCC via the Make system to tell it other places where the pound include files might live, but by default this means in user slash bin slash include, and user include, which you've never looked at before, but they exist. This means, at least in our world, just looking at currently working directory over your compiling, and that's probably where they are, okay. Make sense?

Another thing you might now know about these things is just like pound defines in many ways these are instructions to search and replace this line with something else. This one's easier to deal with because you have a sense of what vector dot H looks like. What this does, when the preprocessor folds over that line and says, "Oh, pound include vector dot H in double quotes. Let me go find it. Oh, I found it." It removes that line right there, and it replaces it with the full contents of the vector dot H file. Does that make sense to people?

And so the stream text that it builds for you as part of preprocessing, the output of preprocessing, it's what's called a translation unit where all the pound defines and all the pound includes have been stripped out. It creates the text that's actually fed to the compiler on behalf of this line right here. It would replace it with the contents of vector dot H as if you'd typed it in by hand there, okay. Does that make sense?

Now you say, "Well, why don't I just type in all the prototypes every single time at the top?" You want to consolidate all the prototypes to one file so that everyone agrees consistently on how all those functions should be called. But if you wanted to, you could just get rid of this, and if you're only gonna use one or two of the functions, you can manually prototype them right there. And as long as it's consistent with the real prototypes that exist in the dot H file, it wouldn't cause any problems, okay.

The pound include process is recursive. So if you pound include a file that itself has pound includes, it will keep on doing until it just bottoms out, okay. It does basically this recursive depth research. It's like random sentence generator without any random numbers, okay, where it builds a full stream of text built out of all the pound include files until it just has one stream of non pound include and non pound define oriented text that gets fed to the compiler, okay. Does that make sense? Okay.

So there's that. If you want to experiment and you want to see what the product of just preprocessing is, what happens when just the pound include and the pound defines are stripped out, go create like a three line file with two pound define constants, and just pound include a dot H file that you write yourself. Don't pound include any system headers because then the output is really, really long.

But if you want to do this, GCC, you're used to seeing something like GCC, the name of a file dash C, like let's say vector dot C or something like that. You haven't typed that in yourself, but you see that published to the screen every time you type make, first time at three and four. Well, dash C means compile, but don't try to build an executable. There's actually something a little more drastic, dash E.

What that means if run the preprocessor and output the result of preprocessing, but don't go further than that. So that means if you look at this file, you'll have some senses as to what it should look like before. You certainly know what it looks like before preprocessing. All of the components that make up this file and vector dot H and anything that vector dot H pound includes will be spliced in sequence to build one big translation unit, okay, with all the prototypes and all the implementations that are in vector dot H, vector dot C rather, to the compiler itself, okay. Make sense?

Okay, as far – what happens if vector dot H pound include – oh, I'm sorry. You know hashset dot H pound includes vector dot H. Suppose I were airheaded and I said, "Oh, I want – you know, I think that vector dot H should also pound include hashset dot H." You could if the preprocessor weren't very smart, and you also didn't have the power to prevent this. You could get circular inclusions. Oh, I better include that. Well, I have to include that. Oh, I better include that. It just could go back and forth forever.

The preprocessors will solve this problem a while ago. We're not he first people to accidentally do that, but you've also seen things like this. If not defined, something like vector dot H, they'd go ahead and define it, and then list all the prototypes that come in vector dot H, and then mark the end region. The very first time that vector dot H gets pound included, or presumably this is the contents of that vector dot H file, as the preprocessor folds over it, it looks in this and goes, "Oh, have I not seen this little token before?"

And if it hasn't, it's like, "Okay, well, then I guess this is safe to do." It'll come down here and define exactly the same thing. You don't have to associate anything with this key right here. It's just basically like a valueless key and a hashset behind the scenes, but as long as it's defined, then if for whatever reason this pound includes, either it's self directly or something that would pound include vector dot H, the second time it's the preprocessor tries to digest it as part of the generation of the translation unit.

It'll come here and say, "Oh, is this not defined?" No, actually it is defined for reasons that may not be clear to me, but I defined it earlier apparently, so it'll circumvent all this and put an end to the vicious cycle, okay. Make sense? Question over there?

Student: Yeah, just a question. The reason why you don't want to include CPP files for that very reason?

Instructor (Jerry Cain):No, actually that's a slightly different reason. All the dot H files, they declare prototypes, but nothing in dot H files ever emits – has any code emitted on its behalf. Like you declare structs, but it doesn't actually generate code in response to that. You're not supposed to declare storage for anything in dot H files except occasionally a very clever way of declaring a shared global variable, okay.

But the dot C files and the dot CC files, they actually define global variables and global functions and class methods and things like that, things that really do translate to zeros and ones in the form of machine code, but we view them as like M of R 1 is equal to R 3 plus 12 or something like that, okay. But dot H files are supposed to be just about definitions that have no cogeneration associated with them so that you can read them multiple times.

Like how many files are there for Assignment 4 and every single one of them probably pound includes vector dot H, right? If they all pound included vector dot C, then they would all be defining vector new and vector dispose, and so when time came to build RSS new search as an executable, you'd have like three or four implementations of the same function. Does that make sense?

Declaring the prototype for a function is very different than actually defining the function. One has code emission associated with it, the compilation actually generates code on behalf of the implementation. It doesn't do anything on behalf of the prototypes, okay.

Student:(Inaudible).

Instructor (Jerry Cain):Yeah, absolutely. You're not required to do this. You just try to choose tokens that are very, very, very unlikely to come up anywhere else, okay. I mean this might be what you choose every time you have a vector dot H file, but presumably, you only have one vector dot H file, which means you'd only have one token defined like this. And when you really use normal pound defines, you just avoid the leaving underscores and the trailing underscores, okay. Does that all make sense? Okay.

So if you get a chance, it takes you all of 15 seconds to do this. Just type in by hand GCC space dash capital E, and then the name of some dot C file in the directory where you happen to be, okay. And you'll just see it like tons and tons of stuff, but toward the end, you'll see familiar codes. You'll see the vector dot C code you wrote at the end of it, but at the top, all the prototypes and any of the dot – the stuff inside the dot H files that happen to be pound included by vector dot H, okay, and also by vector dot C for that matter. Question in the back?

Student: Yes. So you said that that's the way they had them including circulation.

Instructor (Jerry Cain):That's one of the ways. That's the antsy standard way of doing so, yes.

Student: So my question was if that was not included, what did you say?

Instructor (Jerry Cain):Most preprocessors are smart enough that they don't want to commit to circular recursion just because you're not telling it to not do that. Most of them are very smart and they just keep track of it. And I think by protocol it understands that there's no value in ever pound including something twice, but earlier implementations of preprocessors weren't interested in solving every single problem that might come up.

It wasn't – I don't want to say it's an edge case. It's probably a very common case, but in theory, you don't want to just assume that the preprocessor does the right thing, so you just want to make sure it couldn't possibly fail you or infinitely recourse and loop forever, even if you're using like some dummy implementation of the preprocessor, okay.

Some compilers have their own versions of this. I've seen – ten years ago I saw a preprocessor directive called pragma, and it had this optional word over here called once. That was just a more condensed version of trying to do exactly the same thing here without having to invent these names. This doesn't exist, and certainly not antsy standard, and it used to exist in code wear and I don't even see it in code wear anymore.

But different preprocessors can do whatever they want to to extend the standard preprocessor directives. You should just concern yourself with pound define, and if you want if not defines and if defines, and the L's, but really just worry about pound define and pound include. And if you know what those are doing at preprocessor time, then you're certainly walking away with a good amount of information, okay.

So there's that. Let me draw some pictures so you'll have something to write down. So this is vector dot C, and it has this as a code base in it. And it has this file, this file, and this file pound included at the front of it. Let's just say that this is A dot H, and B dot H, and C dot H. I know that's small, but you can just name them anything you want to, okay.

You know that it'll go and find the contents of A dot H and B dot H and C dot H, and as part of preprocessing, what it'll do if the contents of A dot H happens to be that, and the contents of B dot H happens to be this, and the contents of C dot H happens to be this, it really will build a stream of text that's consistent with all these stacked emoticons. This is the stream of text it would build in memory, and the nose list smiley face would be at the bottom. And that stream of text would be passed on to the true compilation base, okay.

Everything that resides in here is still supposed to be legal C, it was just spread among multiple files at this level, so that things like prototype and struck definitions and class definitions and pound define macros and constants could all be consolidated to one place. You're familiar with that concept, right, once used from everywhere, okay.

Well, if you let it got further, it will now compile, okay, where it will take this stream of text as if you typed it in character by character this way and compile it and emit assembly code on your behalf, and as long as there are no errors, it'll build the dot O file, okay. As soon as it finds one error, it'll say, "Oop, an error." And you know, you probably remember the C++ compilers from X code and from Visual Studio C++. When it gives you an error, it gives you a lot of them, and it goes on for pages and pages and pages.

You can suppress it. You can tell it to stop after one error if you want to, but just assuming that everything compiles cleanly, this by default would generate a vector dot O file, okay. And you've seen these dot O files pop up in your directories. This would have all these assembly code statements. If it were compiling to CS107 assembly, you might see things like M of R 1 is equal to SP, things like that, the things that actually emulate the implementations of all of the functions that happen to exist in this translation unit, okay. Does that make sense everybody? Okay.

So what I want to do is I want to talk about compilation and linking kind of simultaneously. And I'm just gonna go through one. I don't want to say it's an easy example. It's actually quite sophisticated, but it's a short program and I can just talk about what happens, and then talk about what happens when you just stop – when you start to remove pound include statements, okay.

Now I am being GCC specific in my discussion of compilation. I'm just doing so because GCC will probably become the most important compiler to you, at least at Stanford, if you're programming in C++, okay. Let me just give you a sense as to what the dot O file would look like in response to this dot C file. Let me just write this file called main dot C. It's gonna be a full program. It's not gonna do anything, but it's gonna be legal C code, and it's gonna cause some functions.

I am going to pound include STDIO dot H. The only thing that's relevant is that it defines the printec function, okay. I'm also gonna pound include STBLIB dot H with the L right there. This is gonna define malloc and free. It also defines realloc, but I'm not gonna call realloc. And I'm also gonna pound include assert dot H, not N, H, and this is the program.

Nth main, nth ard C, car star ard V, it's an array. And I'm just gonna do this. It's like four or five lines. Void star memory is equal to malloc of 400. I'm going to assert that memory is not equal to null. I'm going to print F (inaudible) because if I've gotten this far, then I know that I got real memory, and I'm gonna celebrate by bringing it.

So this is in place just to demonstrate exactly what compilation does. Now pretend we're in a world where there are no other architectures beyond the mock CS107 architecture we discussed last week, okay. So on the CS107 chip, and I feed this to GCC in accordance to the way that the make files that you're dealing with actually would call it. It's going to run it through the preprocessor. You know that these three things would be recursively replaced to whatever extent it's needed to build one big stream of text, which at the end has this right here, okay.

This right here corresponds to that in this emoticon drawing over here, okay. I don't have to generate the full assembly codes for this, but the interesting parts are gonna be this. This is the full dot O file that's generated as the compiler digests the expansion of this to a translation unit. Preprocessing takes this and builds a long stream of text without pound include and pound defines, and that's fed to the GCC compiler that actually generates that O code for you.

You certainly should expect there to be a call to malloc, okay. You would actually see some lines right here like SP is equal to SP minus four. M of SP is equal to 400. Those things should be familiar to you based on what we talked about last week. I'll move over to the right, okay.

You would expect to see a call to printec. You would expect to see a call to free. You would expect to see RV is equal to 0. You would expect to see a return at the end. Those are gestures to the interesting parts of this program from a compilation standpoint, okay. Why isn't there a call to the assert function? Because I included preprocessing in the discussion, and that right there doesn't define an assert function. It declares or defines a way to take this right here and replace it with an expression that doesn't involve an insert function, okay.

There would actually be a – based on the way I wrote it before, I didn't preserve it. Remember how I called F print F before? That's the file star version, or basically the IF stream version of print F. There would be a call to F print F in here as well because of the way I defined assert. Does that make sense? Okay.

So there's that. This is a clean working program. It's not very interesting. It does lots of business and has the weirdest way of deciding whether to print yay or not, but nonetheless, it would compile and it would run. It doesn't even (inaudible) from memory because I'm very careful to free it down here, okay.

Compilation generates this dot O file. If I don't include a flag inside the make file or with the GCC call, it'll actually try to continue and build an executable. By default, it's named A dot Out. If I just use GCC right here, if I want to suppress linking in the creation of an executable and just stop at the creation of a dot O file, I would pass it – I wouldn't call GCC, but I would call GCC dash C. It means stop after compilation, okay. And you've seen the dash C's fly by with all the GCC calls that are generated from make. Make sense? Okay.

If I don't include this, then it will try to build an executable. By default, it would create something called A dot Out unless you actually use the dash O flag to specify what name should be given to the product. And if I say my prog for my program, then it won't use A, its default, (inaudible). It'll actually name it my.prog, okay. The only requirement that's needed past compilation, this is compilation, the generation of this.

When it tries to create an executable, you're technically in what's called the link phase where it tries to bundle all the dot O files that are relevant to each other. In this case, there's only one dot O file, at least exposed to us. And it tries to build an executable. The only requirement that really – you really need is you need there to be a main function so it knows how to enter the program. You have to have a definition for every single function that could potentially be called from anywhere, okay. And you can only define all the functions – each function can only be defined once, okay.

There’s not many link errors that can happen when you’re trying to create an executable, okay. Does that make sense? Now by default it actually links against some libraries that are held behind the scenes that provide the implementations of print F and F print F and malloc and free and realloc and all of those, okay. Does that make sense? Okay.

So there’s that. This is compilation. This is linking right here. I’ll let you say so. And if I type in dot slash my prog, it’ll run this thing, print yeah, and we’ll have a working program here.

What I want to do, I only have a minute, so I’ll just kind of give you like a little teaser as to what we should – what we’ll see on Wednesday. I want to kind of tinker with what happens if I forget to pound include STDIO dot H. All that, that just confuses matters a little bit with regard to the definition of print F. Does that make sense? Okay.

Then I’ll say what happens if I forget to pound include STDLIB dot H and I don’t have explicit prototypes for malloc and free visible? They’re not included in the translation unit, so they’re not around during compilation, okay. What kind of impact does that have on the ability to build A dot L or my prog? And the most interesting of the three is what happens if I accidentally exclude the definition of the assert macro, so that it’s not visible during compilation. Does that make sense? Okay.

Well, I have negative ten seconds, so I’ll let you go. I will talk about those three things. I’ll reproduce this on Wednesday and we’ll spend the first half an hour talking about it, okay.

[End of Audio]

Duration: 50 minutes

ProgrammingParadigms-Lecture13

Instructor (Jerry Cain):There we go. Everyone, welcome. I have two handouts for you today. One project is mid-term and its cousin, the practice solution. I haven't written our mid-term yet, but I'll get to that this weekend. I am typically very good about imitating the structure of the practice mid-term but I never promise that I will. If I come up with some new idea that I think is fair to test the material, then I go for it. Okay, but all of those problems on the practice mid-term are drawn from previous mid-terms over the past three years. It's a little longer, just a hair longer than you're likely to see.

Typically it's two, three or four questions, depending on how intense any one question is. This one has five on it with some short answers. My recommendation, if you can do this – it's difficult to, but if you can take the practice mid-term during some three hour block and just write it out in full and then grade your own work, that's a much better way than just saying, looking at the mid-term looking at the solution saying, yeah, that's exactly what I would have done because it isn't.

Okay, so try and write out the solutions and then you'll have to invest some energy in deciding what types of errors are significant and what types of errors are not. I can tell you right now that we're very concerned with the asterisks and the ampersands and the arrows and the dots and the casts. Ultimately, the questions are really just hopefully interesting back-story and vehicles for us testing all of that stuff and I don't care about the four loops, I don't care about the flux pluses unless it's point arithmetic.

Okay, I really care about the asterisks and the double car star casts and things like that, so don't think that they're going to be viewed as minor errors. They're not, because a car star cast is very different from a car star star cast. The others will cast it to be a struck triple star. It's just as wrong if it's not correct, so concern yourself with those things.

The mid-term is a week from tonight. It is in Hewlett 200. That's probably one of the largest lecture halls on campus. It's not in this building, obviously, it's across the street, so there's that. You can take it any time on Wednesday during some three-hour block that fits in between 9:00 a.m. and 5:00 p.m., if you can't make it that night. You can't start it during lecture hour. If it's best for you to skip lecture hour that's fine, if at all you can make it that night, it just makes everybody's life easier because otherwise somebody has to be around to make sure we can answer your questions. We will be around, there's no doubt about it, but don't take it earlier just because it's convenient for you. Please try to take it at night if you at all can. There's something else. I updated the website yesterday, but I didn't make an announcement because another website wasn't updated yet, but immediately after lecture on Friday, a friend and colleague of mine – colleague is such a snotty word – a co-worker of mine – Facebook is giving a technology talk in actually as it turns out, the same room where your mid-term is so you can practice and get a feel for the room on Friday by going at noon.

I saw him give the talk about three months ago and at the time I was watching him give it I said, this is exactly the type of material, the type of talk that helps motivate the beginning four weeks of CS107. He won't talk specifically in terms of realic and malic and free and vectors and half-sets, but when he talks about the system that's in place to store the billions and billions of photos that they have, and they need laser-fast access to any particular photo that might need to be served up, that's an incredibly difficult infrastructure problem and it relies on this very clever indexing scheme in order to go and find a photo somewhere on one of the – I can't say the numbers, n number of servers where photos reside and it has to do that quickly. And so it's actually concerned with things like minimizing the number of disk reads and things like that and so, very little of it is beyond the scope of what you've learned in the first few weeks of 107. We're also giving you pizza and soda so if you feel like you're sacrificing lunch then you're not really because we're gonna be giving you food. If you're at all interested in going to that, please RSVP by visiting 107 which leads you to the ACM website. Just RSVP so that we know to get you some pizza. It's not televised unfortunately, so you actually have to attend something live if you want to go see this, and we can't televise it, I think, for obvious reasons. Okay, so there's that.

When I left you on Monday, today's Wednesday, when I left you on Monday, I was five minutes into an example that I want to use to illustrate how compilation and linking work, and I'm gonna actually frame it specifically in terms of GCC. This was the program I wrote. IntMaim, I really don't care about those arguments, but I do care about those arguments, but I do care about this, void star memory is equal to malic of 400. I want to assert in the style that we're used to, mem is not equal to nul. I want to print yay, excise m, I want to dispose of the memory and I want to return zero. I'm only allowed to call malic and free without running into any issues. If go ahead and pound include STD, LIB.H, there's a D right there. I only can call Print F cleanly if I pound include that right there and a certain macro is available to me via this header file. This is the entire program, save for the implementation of malic and Print F and free which resides in standard C libraries. If I compile this and I just do GCC, whatever this file is named, it can be made to generate two things. It can be made to generate a .O file and it can be further made to generate an executable. Based on the way this is set up right here, you wouldn't be surprised that there's a call to malic instruction somewhere inside there, that there's a call to Printa, that there is a call to the free function, that eventually there's an RV as equal to zero, and there's a return. There's actually a NSP is equal to SP – 4, up top there's the corresponding SP is equal to SP + 4 right there.

Those things should make sense to you because you know what happens in response to local variable declarations and return values and function calls, okay. That make sense to people? If I were to compile this, even though it's not a very interesting program, I compile it. It generates this assembly code. This is just compilation, this is linking by default, even if you just have one .O file, what linking really is, is it's the blending and merging of this .O file and any other ones you have; but in this case we only have one. The .O file with all the standard libraries and the standard .O files that come with the compiler, the standard compilers have the .O code for Print F and malic and free and things like that. It basically does what is necessary and stacks all of them on top of one another to build one big executable. It usually splices out everything that's not touched and not needed but it includes the code for any function that might come up during execution, okay. It has to be able to jump anywhere possible while a program is running. Because this is set up the way it is, it'll compile cleanly, it will run, it'll print yay as side effects without really making too much noise. It'll allocate a buffer of 400 bytes, make sure that it succeeded and then free it and return zero. It's just this very quick program to really print yay, but to also exercise malic and free and confirm that those things are working, okay. What I want to do, is I want to see what happens if I comment out this right there, just give you some insight as to how little, what little impact the .H file has on compilation and linking.

If you were to say that this would generate compiler errors because it doesn't know about this Print F function, you would be right for some compilers, okay. As a result of commoning that line out there, you know enough to know that the translation unit that's built during pre-processing won't make any mention whatsoever of Printa, okay. So when time comes to actually check to see whether or not Printa was being called correctly, it doesn't have the information it normally has, okay. Many compilers would be like whoa, what are you doing? I've never heard of this function, I don't like calling functions that I don't see prototypes for and some might issue an error; GCC des not. We'll talk about Print F in a little bit, in a little bit more, but if during compilation it sees what appear to be a function call, what it'll do is it'll just infer the prototype based on the call. Does that make sense to people? Okay, it sees that a single string is being passed in here, compilation would issue a warning saying no prototype for Print F found, but it doesn't block you from compiling it still with generated .O file. We are calling it correctly as it turns out, okay. As long as you pass in at least one string, Print F is able to do the job and if there's no place holders in here, so it's not gonna have any problems executing. By default, when GCC infers a prototype, it always infers a return type of Int. Turns out that is the real return value of Printa, we normally ignore it, but the return value is always equal to the number of placeholders that are successfully matched to. This particular call would return zero, but I'm not concerned about the actual return value. I'm concerned about the return type; it happens to mesh up with what's inferred by the GCC compiler. Does that make sense to people?

Now, if several other Print F calls were to appear after this one, they would all have to actually take exactly one argument, because it's inferred a prototype. It's actually slightly different than the one that really is in place, okay. Does that make sense to people? So you may say, okay, it's gonna compile here. What is the .O file going to look like? It's going to look exactly the same. The .H file just consolidates data structure definitions and prototypes. There's no code emitted on its behalf; all it does is it kind of trains the compiler to know what's syntactically okay and what is syntactically not okay, okay. Makes sense? But as it infers this prototype, it's one slight hiccup in the form of a warning during compilation, but it still does SP is equal to SP – 4, it still copies two M of SP, the address of this capital Y, it still calls Print F and it technically expects RV to be populated with the return value, although this happens to ignore it, okay. You generate this and I'd say more than half of the students assume that when you link to try to build this executable, that somehow its gonna break down because it doesn't know the Print F should somehow be included. Do you understand why people might think that? Yes, yes. Every time you try to build an executable using the GCC system, technically you're using

GCC but you're using a cousin of it called LD, which is just for load. I'm sorry, for link. It always goes and searches against the standard libraries whether or not there were warnings during compilation or not. Print F sits as a code block in all of those standard libraries, so it happens to be present during the link phase, even though we never saw the prototype for it.

The presence of a pound include has nothing to do and makes no guarantee about whether or not the implementation of any functions defined in that thing or available at link time. If something is defined in the standard library set it's always available, whether or not we behave with the prototypes. Okay, does that make sense to people? Okay, if I bring this back and I comment this out, then it doesn't have official prototypes for malic or free. So if it comes down here, it doesn't even see that line, it expands this to a bunch to a prototypes and data definitions, expands this to at least the definition for a cert. It's fine with that, looks at this as well, I have no idea what malic is. We're calling this like a function, so I'm gonna assume it's a function; I'm inferring its prototype to be something that takes an Int and returns an Int. It does not actually look how it's being used in the assignment statement to decide what the return type is. So it'll issue two warnings in response to this line. It's inferring the prototype and then you're assigning a pointer to be equal to a bona fide, what's supposed to be a plain old integer, okay. There are no problems with this because it sees the definition of cert, Print F is fine, it looks at this right here and it doesn't like this line either because it hasn't a prototype. It infers one; it issues a warning saying it's inferring a prototype for free. It assumes a, it takes a void star, it assumes it returns an Int, which is not true; but we're not looking for a return value, so it's not a problem, okay, and then comes down here.

So this would also generate the same exact .O file. It would issue three warnings okay, two for missing prototypes and one for incompatibility between L value and assigned value; but it would create this and when we link it, it's completely lost memory. There's no record in here that some .H wasn't around and that there were warnings. It's just that there's a little bit of risk in what's happening here, but I generated it because this as a .O file is consistent with the way you wrote the code here, okay. Making sense? Okay, so it goes on and links and when we run this, it runs beautifully. If I comment these two lines out, then I get a total of four warnings, but it still generates the same .O and it generates the A.L file that still runs. Yeah?

Student:I understand why you said it runs beautifully. It somehow understands [inaudible] right there.

Instructor (Jerry Cain):It does run beautifully, the question is he doesn't understand why it runs beautifully. All it does is it tells the compiler what the prototype are. There's no information in .H files about where the code lives. The link phase is responsible for going and finding the standard libraries. That's where malic and free and Print F are defined. It's not as if there are hooks that are planted in the .O file because of what the pound includes used to be. As long as this and that and that are in there, and they will be whether it's the result of a clean compilation or one with three or four warnings. It'll still have these and so when we link against the standard leverage then set, it'll try and patch these against these three things again. The same symbols that exist in the .O files and it'll create the .O files. Does that make sense? Okay.

Student:[Inaudible].

Instructor (Jerry Cain):That's a different problem, but that isn't the case here, okay. I haven't defined my own Print F or my own free, okay. If I comment these two things out, then I get all the warnings that I've talked about so far. If I bring them back, I have a clean compilation again. If I comment this out, we have a completely different scenario. If it doesn't see any mention of how a cert is introduced as a symbol to the compilation unit, it comes down here it says, I know what malic is. It's a function that takes a size T and returns a void star, okay.

It comes here and it's like I don't know what a cert is, so look at it. If I didn't tell you what a cert was two days ago, or I guess, yeah two days ago, you would say oh, that must be a function that takes a bullion and returns a void. Okay, but we know that it really isn't that because we know how it's officially defined in a cert.H; but because this has been commented out, it looks at this and it's not like the word cert is hard coded into the compiler. It actually assumes that this is a function call. This would now appear in the .O file. Does that make sense to people; yes, no?

Okay, we compile fine, we compile fine, we compile fine, generates this, the link phase would actually fail and the reason it would fail is because even though there's code for Print F and malic and free in the standard library set, they are real functions. A cert doesn't exist anywhere in the standard library set, okay. A cert was just this quick and dirty macro that was defined in a cert.H. Okay, make sense? Okay, so there's that, and I bring it back, everything is obviously restored. The prototypes and I think this is a fairly astute way of putting this; the prototypes are really in place so that caller and callee have complete agreement on how everything above the safe PC in the activation record is set up. Does that make sense to people? The prototype only mentions parameters. The parameters are housed by the space in an activation record above the safe PC. Everything below the safe PC is the callee's business, but when I call Print F and I jump to Print F code, we have to make sure that the callee and the caller are in agreement with how information was stacked at the top portion of an activation record, because it's just this trust where if I'm calling Print F and I lay down a string constant right here, and I say, take over, Print F has to be able go, oh, I just assumed that there's a string above the safe PC. I hope so, because I'm treating it as such and if it isn't there are problems, but if it is, it just prints it. Does that make sense? Okay.

If I were to do this, let me write another block of code. This is actually a really weird looking program, but this is typical of the type of thing you see in 107. Int, Main, I actually don't care about arc c and arc v. I'm gonna declare an Int, I'm just gonna call it num, and I'm gonna set it equal to 15. Actually, you know what? I lied. I'm gonna set it equal to 65 and I'm gonna do this, Int length is equal to, no laughing, stir limb, ampersand of car start, ampersand of num, num, okay. So I'm calling star line in this completely bogus way. I want a Print F; length equals percent D/M. I'll just put length down and I will return zero and let's for the moment not put any prototypes up there. Okay, suppose I completely punt on the pound include section, compiles this, it's like that's fine. Oh, I don't like that, I'm seeing a function call, I haven't seen a prototype for, but because of the way I'm calling it, I'm going to assume it takes a car star, followed by an integer, okay, and I'm gonna assume it returns an Int, because I always do that for prototypes that I make, that I infer information about. The assignment works fine. It prints out whatever length it happens to be bound to, and then returns. So this, if were to compile this, it would only issue one warning. Does that make sense? Now that call is totally messed up. I don't know how often you've had to call star LAN, you've called like mem copy and star copy and all that. This just takes normally one argument, which is a strain and returns the number of visible characters before the backslash zero. That make sense? Okay.

So the way I'm calling this, this is where num resides in memory. I put a 65 there. This is where length resides, okay. I haven't initialized it yet, but it calls star LAN before I can initialize length, so if that's the activation record and it preps for this call right here, and it's inferring the prototype, it goes okay, I don't know what we're doing, but I'm gonna do an SP is equal to SP minus eight. I'm gonna put the address of num right there. It has to be car star, so it thinks that there are characters right there, four of them okay. Make sense? It puts a 65 right there. It leaves SP pointing right there and then it calls star LAN. When this generates a .O file, all it has inside that's of interest to the linker is the call to star LAN. You may think that during link phase that it's gonna freak out because it's gonna somehow know that star LAN only takes one argument. That is not true, there's no information, there's no direct information inside the .O files about how many parameters something is supposed to take. You can look at the .O file and you can see SP is equal to SP minus eight versus SP is equal to SP minus four, versus SP is equal to SP is equal to SP minus 16. You have some sense of how many parameters there are; but not really, because it might be one 16 by struct or four, four by integers, okay. When it does the linking it just looks for the name, it doesn't do any checking whatsoever on argument type, I'm sorry, on parameter type.

So the fact that this signature is zonked and messed up, it's irrelevant to the link phase. All that it looks for during the link phase is for something that responds to the name star LAN and that's exactly what happens. So when this executes and it jumps to star LAN, star LAN inherits this picture. This is where our safe PC is set up. The real implementation of star LAN was written and compiled to only have one car star variable. Does that make sense to people? So its assembly code only goes this high. It may actually detriment this by some other multiple of four bytes for its local variables, okay. It does a four loop inside really, is what, all it does and then returns it to here. But do you understand why the 65 is more or less ignored? Does that make sense to people? Okay, so as it turns out, this will compile with one warning. If I want to turn off the warning, I can do this. I can actually manually prototype it at the top. That's the type of thing that comes into the translation unit as a result of preprocessing anyway, so if I want to suppress the warning there, and then just manually prototype, because obviously I think that's the prototype because that's the way I'm calling it. Then I can just do that okay.

A lot of times you will see people only include the prototype manually. The alternative is to pound include this big .H file that's closed on compilation and it you're concerned about, if you have to remake a system 75 times a day from scratch, and the number of pound includes actually impacts compilation time, a lot of times you'll just manually prototype things instead of pound including everything. It's a little risky, because technically – less egregious, but technically incorrect versions of this mistake could actually happen, and cause code to compile that probably shouldn't. But this will now compile cleanly, because I said this line is perfectly good as is. When it runs, it calls star LAN and only thinks about everything below that little arc right there, okay. It actually treats that as a car star, it has no choice but to do that, we even coached it to think that it's a car star for the call. It's gonna return, it's gonna bind length to some value. It's gonna print it out. Then any idea what's gonna be printed out by this program? Yeah?

Student: Zero?

Instructor (Jerry Cain):It would be zero on basically most systems; I'm gonna say most, many systems. It would actually print out one on some other systems.

Student:[Inaudible].

Instructor (Jerry Cain):I know, it does and it actually doesn't have anything to do with the nul character. I'm assuming that the four byte integer that really resides here is stored this way. Does that make sense?

If that's the way it's set up and there really is a single byte of zero all the way to the left and so as far as this argument is concerned, it actually thinks it's pointed to the empty string in this little static space on the stack okay. Make sense? That is the big NDNGO. In the little NDN system, these would be reversed, right? And so 65 happens to be a capital A. It doesn't even care that it's 65; so much of it is non-zero. So on little NDN systems, this would actually print out length is equal to one, okay. Does that make sense?

The interesting part is that this is a complete hack. I mainly prototyped it right there. Try not to do that, because it allows things like this to happen. It turns out it doesn't cause any problems. I'm sorry, it doesn't cause any run-time problems, it'll actually execute properly and because this happens to point to either 000 65 or 65 000, both of those can be interpreted from the beginning of that sequence, as some c-string. One happens to be the empty string, the other one happens to be the capital A followed by a backslash zero, okay.

It will just have some response, even though if it's a little weird, okay. Make sense? Yeah?

Student:[Inaudible] the second parameter?

Instructor (Jerry Cain):The caller does not. The caller actually places it there, but think about the implementation of star LAN was really compiled with the normal prototype, not this one. So it only reaches four bytes above, at most four bytes above the safe PC, which is why I draw this arc right here, okay. Does that make sense?

This still sits there; it's still popped off the stack when star LAN returns because this did an SP is equal to SP minus eight. It just assumes that 65 was integral to the implementation or to the execution of star LAN and it pops it off afterwards, okay. Does that make sense?

That was – it wasn't my exam question, Julie Slonski gave us like 12 years ago. I couldn't believe it when I wrote it. I was like wow, it was really hard. I don't know whether a couple of people got it right or not, but I thought it was very interesting that's why I wanted to put it in lecture.

Let me give you the opposite scenario here, though. Suppose I do this, Int, mem compare, void star, v1 and I just mess up and for whatever reason, I think that somehow a mem compare only needs one void star and then I have some block of code that declares an Int called n is equal to 17 and I do Int m is equal to NPMCM, ampersand of n, and that's it, that's all I care about.

Okay, it's not a very interesting block of code, I just want to see what happens if we call a function with only one argument that really expects three, okay. You may not remember the real prototype for mem compare. It's used incidentally in assignment four, but it's kind of like star comp, except you explicitly pass in the number of bytes that should be compared. That's why zeros are meaningless to a memory compare. The actual prototype is this and this is the way mem compare was compiled behind the scenes, v1 void star, v2 Int size. That's the real prototype. The call here would declare m, stack it right there with the 17, it would put an m there that has no value yet. When it calls mem compare according to that prototype, there's a safe PC right there, there's the address of n is right there, and this is compiled with the idea that only the four bytes above the safe PC actually are relevant to the implementation of mem compare. Does that make sense to people? When we transfer execution to the real mem compare that has a completely different idea of what the parameter list looks like, it inherits that as a safe PC and it's like wow, I have 12 bytes of information above the safe PC. So it just accesses them, okay, so this overlays the space for v1. This uninitialized value overlays the space that's used for v2 and it happens to inherit 17 for the size and it executes according to those three values. I'm not saying it's sensible, I'm just telling you what happens, okay.

So because I did that and because this is the only thing that's part of my main function, it would compile, it would execute, it probably would crash, okay. It's probably the case because this is uninitialized, so it's very unlikely that it has as a random four byte address inside of that point, something that's really part of a stack, for the heap, for the code segment, okay. Does that make sense? If it happens to be that, then it would run somehow, okay, but it probably would not. You guys get what's going on here? Okay. It's, c as a language, it's very easy to get something to compile, and it sounds like, I mean, you may not believe that, but coming from c plus, plus LAN and 106b, maybe you do realize it now. You're certainly seeing things compile in assignments three and assignments four that are wrong, okay. They just crash so they don't work. If it were a fully, strongly type system where there was no idea of exposed pointer or void star or casting, you'd have to actually get a lot more things right before things compiled. Does that make sense? You're used to templates from C plus, plus and you don't use any generics with void stars and c plus, plus. I'm sorry, you just usually don't.

Even though it's a pain to get things to compile in C plus, plus, it's rarely the case that you crash, okay. To the extent that you use templates, template containers in c plus, plus, you're that much less likely to deal with pointers and so you just don't see as many crashes when you run a c plus, plus program, okay. You do see a lot of crashes with c programs; we all believe that now, okay. It's almost like c plus, plus as a language, the compiler is like this hyper-persnickety wedding planner, where everything has to be in place before it'll let the wedding happen, okay. Does that make sense? Language like c, it's like it'll all work out and so that's what the wedding planner is saying, it's like yeah, it's a void star, it's a void star, yeah, as long as you know what's going on, I trust it'll all execute and if it doesn't, well then, that's your problem, okay. It's really what the c compiler is really viewing it as. This is definitely a c compiler, an exploit right here. You couldn't do something like this in the pure c plus, plus extensions of the c language, okay. Does that make sense? Do you know how you can overload functions in c plus, plus? You can actually define a function with the same name as long as it has different parameter types. You can even have the same number of parameters as long as it can disambiguate between two different versions based on the signatures of the two calls and the data facts of the call, it'll let you define one. You can't do that in pure c. If I have MEMCMP as a function name, then I can only use it once.

What c plus, plus does is very clever. When it compiles it, it actually doesn't tag the first instruction of a function with just the name of the function, it actually uses the name of the function and the data types in the order they're presented on the argument list to assemble a more complicated function symbol. So something like this in c plus, plus would be a call to MEMCMP. I'll write it this way. This is the way it would be set up in pure c. In c plus, plus, it has to be able to disambiguate between multiple versions of MEMCMP with different signatures. So it actually does this, and I'm making this part up, but something along those lines okay. Does that make sense? So if you were to compile this with a c plus, plus compiler and you were to compile that implementation with a c plus, plus compiler, this would be call to MEM compare, underscore void star, underscore P, whereas this would be tagged with MEM compare underscore void, underscore P, underscore void, underscore P, underscore Int. Does that make sense? You might as well call the functions x and y as far as the c plus, plus compiler is concerned, okay. So a call to this from a c plus, plus .O file to this, would lead the linking problems, okay. Does that make sense to people; yes, no? So c plus, plus is a little safer in that regard as well.

Okay, so there's that. I have a few things I want to do. I'm gonna spend today and Friday easing up a little bit before – I've actually formally covered everything that I'm gonna cover on the mid-term. In fact, I'm not even gonna test you on this stuff on the mid-term. I'm just trying to give you a very big picture of what the entire effort is into building a c or a plus, plus executable. I want to go back a little bit and talk about debugging and in particular, give you some lighthearted examples as to why programs crash the way they do. It's one thing to say they crash, that's not very interesting and insightful. Yeah, it crashed, well of course it did, it's c, but why did it crash? It's like why did it crash and what happened at run-time to actually prompt the crash or the segmentation fault or the plus error. I know you have no idea what the difference between those two is. I'll tell you right now what they are, but I just want to show you why programs run the way they do when there are little bugs in there okay, and I mean, if something survives compilation and survives linking, why it runs and it can still go astray.

Let me quickly talk about the two very harsh alerts that are thrown when your program crashes. You're used to seeing segmentation faults and you're used to seeing bus error, okay. You've probably seen seg faults more recently in assignments three and assignment four more. I wouldn't be surprised if you saw a lot of bus errors in assignment two, okay. This right here always comes into plays when you dereference a bad pointer. That turns out to be the case with this as well, but there's different reasons in each scenario. If you ever try to dereference the nul pointer, if you really try to do this, well that wouldn't compile, because you can't dereference a void star, but conceptually, if you were to actually try and jump to the nul address to discover an integer or a car star, it issues a segmentation fault, okay. The reason that's the case is because for the 12th time this quarter, I'm drawing all of ram and I'm drawing the stack up here and I'm drawing the heap right here. Stack, heap, here's the code segment. There's also the data segment is usually actually done here, but I'll draw it up here because there's room. The nul address corresponds to that. Do you understand that the four bytes at address zero, one, two and three, they're not part of any segment, okay. The operating system can tell that. It's like, okay, you're dereferencing the nul pointer. I'm not mapping the zero address to your stack or your heap or your code segment. So I know this can't possibly be right, because you're not dealing with an address that should be under your jurisdiction. It's not the address of a local variable; it wasn't an address that was handed back from malic, so why are you dereferencing it? I'm gonna scream at you in the form of a seg pull, okay, and that's what a segmentation fault is. It's your fault for not mapping to a segment, okay. This is a little bit different. Bus errors are actually generated when you dereference an address that really is somewhere in one of the four segments, but it can tell, or it thinks it can tell that the address couldn't possibly correspond to what you think it is. If you have an arbitrary address here, void star at VP is equal to whatever is equal to and then you do this. If VP really is an address that corresponds to – that resides somewhere on one of the four segments, you will not get a segmentation fault because you are hitting a segment.

Because it wants to make things simpler, compilers adhere to what's called – adhere to a restriction that's imposed by the hardware and the operation system that all integers actually begin at addresses that are multiple of four. That all shorts begin at even addresses, there's no restriction on characters, but basically just to keep things clean and to kind of optimize the hardware, it always assumes that all figures other than shorts and bytes reside at addresses that are a multiple of four. I don't know whether you questioned why I had this random padding every once in a while in the data images from assignment two. Like I said okay, and there's a two byte short that follows a backslash zero, unless the name of the actor is even in which case there's two backslash zeros. That's because I knew you wanted to dereference some pointer in there as a short star and if it happened to be an odd address, even though the two bytes that are there really do pack in a short, the hardware will like whoa, bus error, I don't like that. I don't like you dereferencing odd addresses and thinking that there are shorts there because I know that the compiler would never normally put a short at an odd based address. Does that make sense to people? Okay. If I do this, and so let's say that this right here, if VP is equally likely to be any address that's inside a stack, the stack or the heap or the data segment or the co-segment, then this would throw a bus error with 50 percent probability. Does that make sense? Okay, if it doesn't throw a bus error, then it really does write a two byte seven somewhere.

If I were to do this, then that would throw a bus error if VP was really part of some segment somewhere, but VP wasn't a multiple of four. The address 2002, no it wouldn't put an Int there, right. So it's not gonna let you start laying down a four byte integer at what appears like an odd address, even though it's not really odd. Address 2000, it's fine. Address 2004 and 2008, they're great; 2002? No. 2001? Doubly no, okay. All of those intervening addresses could not house or be the base address of an integer. It's like a block where all the houses have to begin with like, have a, to be a multiple four, an address. Okay or one of those snobby neighborhoods where everything's like 100 or 200, okay, the addresses on the houses. Does that make sense to people? Okay, so when you see a bus error, and – I'm sorry, when you see a seg fault, it's almost always because you have some nul pointer. In theory it can be any address off of a segment, but it's gonna be either dereferencing a nul or a four or eight or some very small no pointer relative address, okay. Bus errors I see less often. It only usually happens when you're dealing with manually packed data like we say on assignment two and you're trying to rehydrate two byte and one byte – four byte figures from arbitrary addresses internal to some data image, okay.

Okay, what I want to do now is now that you have that, you have some more information and when you see bus errors and seg faults at least you have some idea of why, what's happening. Let me throw some code up on the board, and I want you to understand why this program does what it does. Here is the entire program. I'm not gonna concern myself with pound includes, just assume anything that needs to be pound included is, and I'm gonna declare an Int I right there. I'm gonna declare an Int array of length four below it and then I'm going to deliberately mess up. I is equal to zero, I less than – is it less than or less than or equal to, I don't know, I'm gonna include more, that's probably safer and I'm going to do array of I is equal to zero, okay. You see the bug, okay; you see that it's overflowing the bounds of the array. What you may not be sensitive to, and this is something you can only understand after you see a mock memory model, which is actually really close to the real memory model of most function column return mechanisms, is that this code executes with this image in mind. Here is the safe PC of whoever called main. It's actually a function called start that calls main and starts responsibility is to pass the command line to build the Arc V array of strings, count the number of strings that are there, put that number in Arc C. It actually puts a nul after the very last string, so in case you want to ignore Arc C you can, but then this is that array of length four.

This is I, and let me just be obtuse about it and just trace through this, even though you know exactly how it's gonna run. It's gonna set this equal to zero, it's gonna pass the test, it's gonna put a zero right there, it's gonna come around and promote that to a one. It's gonna pass the test, so it actually lays a zero right there. It succeeds in making this two and then three and intermittently getting that right there. After it writes that zero, it promotes this to a four, okay, and you're like okay, something's gonna happen and it's probably not good. It comes over here, it passes this test, so it says okay, I guess –and it's not even gonna say I guess, it just does it, it's not suspicious. It comes over here and it writes a zero to something that's four integers above the base address of a write. So where does it write that zero? Over the four. So it just does that, it comes back up here and it's like wow, that's weird, I thought I saw a four here before, but I guess it was a zero, so I'm gonna promote it to a one and I'm just gonna write a zero over here. Wow, it's a zero already, what a coincidence, and it's gonna keep on doing this, okay and it's gonna basically start right there, go up here and it's gonna keep on cycling here. How long? Forever, okay, and that's because of the way that everything happens to be packed in memory, that this buffer overflow, technically that's what it is, really does damage. It doesn't actually – it kills data. In this case it happens to get a program to run forever, so this is a slightly more complicated version of Wild True, okay. Does that make sense to people? Okay.

There are other variations on that, let me just do a couple of other things here. Let me just assume – I have to change just one line here so I'll give you a second to recover. Suppose I just do this, short array, so the picture, the stack print picture actually changes a little bit. Now the stack picture looks like this, I is still this big fat Int, but now there are four shorts packed here, okay. Makes sense? On some systems this is gonna work fine, fine being a relative term and meaning not badly and on some of the systems it's gonna run forever, okay, for very much the same reasons except there's a little bit of a size indifference thing here that we have to worry about. This is set to zero, lays a zero down there, this is set to one, lays a zero down there, this is set to a big fat two, lays a zero there, three, puts a zero right there.

Then this thing is promoted to a four. I'll put it right there, we'll say it's a big NDN system, okay, so it's really dot, to dot, to dot like there. So if it's a big NDN system, when it overwrites the bounds of the array, all it's doing is it's overlaying zeros where zeros already were, so I'm not saying it's correct, but you actually don't see the problem and it just runs and it takes 20 percent longer than it should have, but you don't deal with things that fast, so you don't really care and it runs and it returns and you think all is great and so you move from the [inaudible] and it's 11:59 and you're like oh, I better test it on the pods. So you bring it over here and you're like wow, it runs forever. Why? Okay. That's because the four was over here on the little NDN systems, and the pods run Linux on X86 machines, they're a little NDN. It writes this for – overwrites this four zero, a two byte representation in little NDN form of a four, with zero zero, which is zero in both big NDN and little NDN and then goes through the same confused cycle that the Int array version did. Does that make sense to people? Okay, so there's that. I have one minute, let me just give you one other example, I have one minute and 20 seconds, I can do it. I actually gave this about five years ago on a mid-term. I thought it was so clever and they didn't. So I had this as a function, void fu and I was curious as to what happens when you call fu. I did this, Int array of size four, Int I and then I did this. Don't question why I'm doing it; I less than or equal to four. The error is the same, the four loop issue was exactly the same, but I do this and the fact that array is not initialized, that's a weird thing to do to an uninitialized array slot, okay. Notice that the array is above I this time, okay, so I'm back with all integers.

There's my I variable, here are the four integers that are part of a larger array, okay, and so it does this and all it does here, this goes from zero up to four, it just demotes this by four, whatever happens to be in there, and because it's allowed to go one iteration too many, right, whatever happens to be here is also decremented by four. Now we know whatever happens to be there, if that's the only set of local variables that I declare, this is the safe PC, okay. That make sense to people? So the safe PC without really knowing, somebody took the safe PC and decremented it by four, numerically by four. What that means is that this as a pointer, which used to point to the instruction after call to fu, something that let's this continue, somebody said I'm gonna make you unhappy and put you right there. The impact is that this thing returns and the callee wakes up and execution carries forth where the safe PC says it should carry forth from. Somebody move the piece of popcorn back four feet and so it says oh, I have to call fu again, and it does and it returns like oh, I have to call fu again. Okay, just that's exactly what happens, it keeps putting the address of this thing down here, but because of this buffer overflow, it actually keeps decrementing the safe PC back four bytes, which means it marches it back one instruction, so how you get this more interesting version of infinite. It kind of is infinite recursion. Right at the end of the fu call, really, really toward the end of the fu call, it calls fu again, okay. Does that make sense?

Okay, so come Friday I will talk to you about Print F and a couple of other things and I will talk about – that's probably it. It'll probably be a nice, easy lecture on Friday. Have a good night.

[End of Audio]

Duration: 52 minutes

ProgrammingParadigms-Lecture14

Instructor (Jerry Cain):We're on. Hey, everyone. Welcome. I don't have any handouts for you today. I actually want to finish up the implementation section of the course. I think we'll get through it today. I'll make it a point to finish it today. Come Monday, we're gonna start talking about multithreading and I'll even preface that a little bit today, provided I don't run out of time. Remember that your midterm is Wednesday evening over in Hewlett 200, the largest room on this end of campus. It's 7:00 to 10:00 p.m. It's open note, open book. You can bring print outs of your programs, whatever you need. Just plan on taking the exam in the three-hour period. It's designed to be more than enough time for the midterm. Okay. There's also the technology talk right after this class right here over in Hewlett 201. So you can first visit Hewlett 200 to see what room it's like for the exam. But in 201, there'll be a technology talk from 12:00 to 1:00. Okay. I left you last time with this example, but I got several questions about how it worked, which probably means that I rushed through it in the last five minutes of class, which is probably true. We had something like this, where I declared an int array of length four, and int i to serve as four loop index, and then I'm just gonna go and do this. I don't care that the array hasn't been initialized. I want to go ahead and do this right here. And then just return. What you probably do remember from Wednesday is that given R memory model, that this would prompt the program to run forever. Why is that the case? Based on this local variable set, we're dealing with this as an activation record. One, two, three, this is the array. As far as that four loop is concerned, it's just one too small. This is the i variable. It goes through and it demotes all of these variables by four. What values were they before? We have no idea. But it will certainly take whatever values happen to reside there and demote them by four. Unfortunately, because the test is wrong, it goes up and it demotes this numerically by four, as well. Now that four isn't so much a four as it is – it isn't so much a negative four delta as it is a negative one instruction delta, because you know that this is the Safe PC in our model. This was supposed to be pointing somewhere in the code segment to the line that called the Fu function. This is planted down there in response to that assembly code statement. Does that sit well with everybody? Okay.

When you do this, you inadvertently tell the Safe PC that is wasn't pointed to this instruction, but that it should back up four bytes, which happens to be this nice round number, as far as assembly code instructions are concerned. And to stop pointing there, and to point there, instead. So when this as a function returns, it jumps back to this right here, and it executes the call, having no memory whatsoever that it called Fu like 14 or 15 assembly code instructions ago. Okay. Does that make sense to people? So this is this very well disguised form of recursion. You won't be able to emulate this on the Solaris boxes or on the pods or the mist, because their memory model is a little bit more sophisticated than ours. They don't put the Safe PC right next to this right here. But nonetheless, that is probably the fifth example of infinite recursion we've seen in the last two days. Let me show you another program. No infinite recursion in this one. I want to write a simple main program, int name. I don't care about the parameters. I want to do this. I want to do – let's just say, "declare and init array." Now, I'm not declaring any local variables whatsoever. So you should already be a little bit suspicious as to what that type of function should do. But imagine the CS 106a program in week four, learning C instead of Java. And they just don't have arrays and parameters passing down. So they have this right here. And then afterwards, they have this things called "print array." And that is it. Now unless there's global variables involved, there's no legitimate, even though we don't like globals, it still would be a legitimate way to communicate information between function calls. But suppose there's no globals, either. The program doesn't think they need globals because they do this: void declare and init array.

And they just declare an array of length 100. They declare the forward variable i. They're not going to overrun the bounds this time. They're gonna get it right. And they're gonna set array of i equal to i. So that's a beautiful little code, but it does very little outside of the scope of the declare and init array function. But for whatever reason, they've decided that they want to declare this array, locally update it to be a counting array, and then leave. Okay. Then they come back and they want to print out that array and this is not that uncommon when we taught in C. They think as long as they name the array the exact same thing, that all of a sudden there's a relationship set up between that array and this one, as if the word array has now been reserved throughout the entire program. And they do this and then they do this for i is equal to zero. They get the forward bounds right again; that's not the issue. Print out percent D backslash N array of i. And they come in during office hours because there's some part of the program beyond "print array" that's not working properly. But then a TA looks at this or I'll look at this and say, "Oh, this is wrong right here." And they're like, "No, that's working fine. It's actually down here." Well, it may be something that's wrong down here, as well, but clearly there's something wrong going on right here. However they make the argument, "Well, it's working." And the answer is, it is working as far as they're concerned, because that manages to print out zero through 99, inclusive. You all probably have some sense as to why that's happening. Try explaining that to someone who's never programmed before. This right here, built-in activation record of 104 bytes, goes down and it lays down the counting array in the top 100 of the 101 slots. Returns.

It calls a function over there that has exactly the same image for its activation record. So it goes down and embraces the same 404 bytes. This is me embracing 404 bytes. Okay. And it happens to print over the footprint of this function right here. It's not like when this thing returned it cleared out all of those bits and said we got to scramble this so it doesn't look like a counting array. It doesn't take the time to do that. So basically, what happens is if this is the top of the activation record and this is the bottom of the activation record, SP is descendant once, this is filled up with happy information, comes back up after the first call returns, comes back to the same exact point, the happy face is still there. We print over it and it happens to be revisiting the same exact memory with the same exact activation record, so it prints everything out exactly the same. Makes sense? Now, there's some advanced uses of this, where it really does help to do this. A lot of times, not in sequential code like we're used to, but about 11 years ago, I had to rely on this little feature when I was writing a driver for a sound card. And you had to actually execute little snippets of code with each hardware interrupt. And so a lot of times, you had relatively little to do in one little heartbeat of an interrupt, and you had a lot to do the next one. In fact, you had so much to do that you weren't sure you were going to have time for it consistently. So a lot of times, you would prepare stuff ahead of time and put it in exactly the right space so you knew where it was the next time without having to actually generate it. Do you understand what I mean when I say that? Okay. So you almost think of this as this abusive, perverse way of dealing with a global variable. And you're setting up parameters for code that needs to run later on. This example wouldn't exactly work that way. This isn't a great example of that, but at least it highlights the feature.

This thing called "channeling," that's exactly what this thing is when really advanced C++ programmers take advantage of this type of knowledge as to how memory is laid out. As far as the problem with hand is concerned, you feel helpless, you try to explain, for instance, if you've just put a "call to printf" right there, that its activation record has nothing to do with these activation records. So it goes in and garbles all of the sacred data from zero to 99. And their response is, well, just don't do that. And comma is out and they think they restored program, but they really have not. Does this make sense to people? Now, I want to revisit this printf thing, actually. I don't know how many of you know the prototype for printf. You really are just learning it, basically, on the fly in context when you're dealing with a simon four and you're just seeing lots of printf's in the start code and you just kind of understand that there's a one-to-one mapping between placeholders and these percent D things or percent G things and the number of additional parameters. Well, you know sample calls are things like this: printf hello. And that's in contrast to C out less than less of hello is string constant with an "endl" at the end. When you have something like this: percent D plus percent D equals percent D backslash N, and you want to fill it in with four and four and eight, if you wanted to do it that way, you certainly could. But the interesting thing from a programming language standpoint is that printf seems to be taking either one argument or four arguments, or really, it takes anything between one and, in theory, an infinite number of arguments. It's supposed to be the case that those things line up, in terms of placement and data type, with the additional parameters. What kind of prototype exists in the language to accommodate that kind of thing? Well, we always need the control string.

That's either the template or the verbatim string that has to be posted to the console. So the first argument to this thing is either a char star or a const char star that can polish supports const. So I call it "control." But then, there's no data type that really has to be set in stone. There doesn't even have to be a second argument, much less a data type attached to it. I could put a string there. If this is a percent S, a float there if this is a percent G, things like that. The prototype for that is dot, dot, dot. And so forth. Whatever they want to type in. And the complier is actually quite promiscuous, and what it will accept is arguments two, three, and four, provided that dot is there. If you don't want to insert anything, it's fine. If you want to insert 55 things, it's great. The complier is not obligated to do any kind of type checking between this and what this evaluates to. There's nothing implicit in the prototype right there. It's just a char star, free form char star, and then whatever you want to pass in. You want to pass in structs? Great. Pointers, structs, you can do that. GCC, for quite some time, has an extension to the C spec that is implementing. Where it does try to do type checking between this and that right there, if you mismatch, it's doing a little bit more work at compile time than it's really obligated to do. It wants to make sure that this printf call works out. So if you were to put percent

S, percent S, percent S, and put four, four, eight there, most compliers would just let you do it and run and it would just lead to really bad things. But GCC will, in fact, flag it and say, you didn't mean to do that. Okay. Does that make sense? This return type – I mentioned this on Wednesday – this return type is the number of placeholders that are successfully bound to. So as long as everything goes well, it would be zero for this call and three for that call. It's very unusual for printf to fail. Scanf, the read in equivalent of printf, can fail more often.

But if, for whatever reason something goes badly, this would return negative one. Do you know how IF streams set the fail bit to true so that when you call the dot fail method inside C++, it'll basically evaluate to true and that's the way you break out of a file reading program? Well, you're relying on the equivalent of printf's return value, which is called "scanf" or "f scanf" to return to negative one when it's at the end of the file. The reason I'm bringing this up is because, based on what we know and the way we've adopted a memory model, I now can defend why we push parameters on the stack from right to left, why the zero f parameter is always at the bottom and the one f parameter is always above that. Let's just consider that call right there. The prototype during compilation just says that either of these calls is legitimate, but when it actually complies that second printf there, it really does go and count parameters and figures out exactly how many bytes to decrement the stack pointer by for that particular call. So the way that the stack frame would be set up is that this would be the Safe PC that's set up by the call to printf. Above it would be a pointer to the string, percent D plus percent D equals percent D backslash N. And this would have a four, this would have a four, and this would have an eight. The activation record for the first call would just have this many bytes, and would have a pointer to the hello string. So the activation records, the portion above the Safe PC actually is completely influenced, not surprisingly, by the number of parameters that are pushed onto it. Now, when we actually jump to printf, and this is where the SP is left, it doesn't have any clear information about what resides above the one char star that's guaranteed to be there. Does that make sense to people?

So really all that happens – I'm gonna draw this arc again, like I did before – it knows about that much, if there were special directives in the implementation of printf that allow it to manually crawl up the stack. But a number of arguments and the interpretation of the four-byte figures that reside there, it could only figure that stuff out by really analyzing and crawling over this control string character by character. This is almost like the roadmap to what resides above it in the stack frame. Does that make sense? So the printf function really does need to get to the control string, and it reads it character by character, and every time it read a percent D – let's say if reads a percent D right at the front, it says, oh, the four bytes above the control string must be an integer. And then it sees another percent D along the way. It says, above that there must be some other integer. And this is how it discovers the stuff that should fill in the control string. So you understand that, otherwise, if it didn't have this right here, this would truly be a big series of question marks. It's still kind of is a series of question marks. If this is the wrong roadmap, if I do percent D plus percent D equals percent D and I pass in three strings, these things will be laid down as char stars. That's what the caller would do. And then it would interpret them as four-byte integers. So whatever addresses happen to be stored there would be taken as unassigned integers, and it would just fill those three things in that way. Makes sense? This is consistent with the way we push parameters onto the stack, from right left, last argument first, then the second to last argument below that, etc., so that the zeroth argument is at the bottom.

Imagine the scenario where the Safe PC is addressed by the stack pointer, but you have the mystery number of bytes below the control string. And that question mark region would be of height zero for the printf hello call, and of height 12 for the printf four plus four is equal to eight call. It would have no consistent, reliable way of actually going and finding the roadmap as to how to interpret the rest of the activation record. Does that sit well with everybody? So as long as you understand that, then at least you have a defense for why that dot, dot, dot – because of the dot, dot, dot, the C spec more of less has – I guess it doesn't have to, but it just made sense to for compliers to implement this left to right parameter pushing strategy. Because they want to support that dot, dot, dot in the language. C++ has to do the same thing, because it's backwards compatible with C. Java just recently introduced the ellipses – I think it was either Java 1.5 or Java 1.6, I'm not sure – but very recently they introduce the ellipses. And so I just know, without actually reading anything about it, that they have to push their parameters on the stack in exactly the same way. Pascal, old school, wasn't old school for me when I was in college, but it's old school for everybody here. I didn't learn Pascal. I learned C first, but when I read about Pascal, it doesn't have this ellipses option. You have to specify the number of arguments. It happens to press the argument on in the opposite order. And it doesn't cause any problems. They had the flexibility to do it an either order because they never had to deal with this question mark region in a struct. Does that make sense? But as far as structs are concerned, you may ask – this is a hard point to make; I'm gonna try and do it. Struct Fu, let's say I have an int code and I have, let's say this. Int code, and I'm just gonna do that right there.

It's unusual for you to have a struct around one data type, but I'm gonna do it anyway. And then I have struct type one, which has an int code and, let's say, several other parameters. Maybe it's the case that the code inside a type one struct is always supposed to be one. So just take this as a series of equipments of the example. If I have struct type two, with an int code at the front, I might require that all instances of type two actually have a two at the front. These are really esoteric examples. I'm just making this up. I haven't done this is past quarters. But this right here is a data structure that I guarantee, whenever I give you a pointer to a struct base, the idea is that there is one of two values that sits right there. It's almost like it's a little opt code in the assembly instruction, in a sense, it to figure out how to interpret what resides, or figure out what resides above the one or two in memory. You could cast that pointer to a struct base knowing that there's gonna be at – or maybe it is typed to be a struct base, which means that you know that there's some kind of opt code or type code sitting there. And then based on the result here, you can either cast this arrow to be a type one star or a type two star to figure out how the rest of the information is fleshed out. A complicated example, but there are various structs – I don't know whether you've looked at – I don't think I've exposed the code for all the networking in the URL connection stuff. If I did, you would have hated Assignment 4 even more, because you would have thought you were responsible for it.

But there are lots of structs that are afforded by GCC and G++ to help manage networking, and I tried to insulate you from that. Old school networking deals with four byte representations of IP addresses. They realized about 15, 20 years ago that they were going to run out of IP addresses pretty soon, so there's actually a six byte universal version of IP codes. It's standard, but it's not really widely adopted yet. There are two different structs associated for the two different protocols. The four byte version IP, version four, there's IP version six. The IP version four struct has been around for some 25, 30 years. Those things aren't going to go away. So when they designed the IP version six struct, they had to make sure that the first half of it had exactly the same structure as the IP four version, and then they extended it with all this extra information. Does that make sense? Do you always know that you're getting a pointer in networking code?

You always know you're getting a pointer to one of those two structs, and you analyze the first few bytes to figure out whether or not you have an IP four struct or an IP six struct. Does that make sense? The reason I'm mentioning this is because now it kind of gives you some sense as to why the first parameter, or the first field in a struct or a class actually has to be at the lowest address. It doesn't have to be, but that's just the way they did it, because they had these types of things in mind. If this and this and this were always the last thing, or the thing at the top of the struct in the activation record, it would always be at a mystery distance from the base address of the entire thing. Does that make sense? Now you could just argue that you could make the code the last thing in a struct so you could do it the other way. It just makes sense to people who designed the spec or designed the compliers. I don't think it's part of the specification, although maybe it is, that the first field is always in offset zero, and you can exploit that knowledge to do clever things with C and C++ structs. Does that make sense? So I'm cranking on time. So what I will do now is I will give you a little bit of a head's up on how we're gonna transition things. Everything so far in C and C++ has been sequential. And I'm talking in Java it's – actually, not in Java necessarily, but all of the C++ and C you've done, in all 106b and 106x, and for example, 107, has either been strictly or seemingly sequential. And you know what sequential means now, because you're waiting for the sequence of RSS news feeds to all load over a five-minute period while you test. So you know what sequence means more than you did a week ago. I want to introduce the idea of how two functions can seemingly run at the same time. That's going to be a huge win in the context of Assignment 4. You'll all be delighted that you're going to revisit Assignment 4 before Assignment 6. And you're going to make it run oh so much faster by using this thing called threading. What I want to do is I want to talk about how two applications on a – and just give you very high level stuff – two applications can seemingly run on the same processor simultaneously, and then use ideas from that to defend how two functions within a process can seemingly run simultaneously. I'll frame this this way. Let's just think about one of the pods. This is the virtual address base, virtual meaning the allusion of a full address base that make has wireless running. You don't think about Make as an application, but it certainly is.

It's designed to read in the data file that you call a Make file to figure out how to invoke GCC and G++ and the linker and purifying and all of those things, to build an instrument and executable. This has a stack segment associated with it. All the local variables of the thing that implements make go there. There is the heap. There is the code segment. While make is running, it's probably the case that GCC, as an executable, is running several times, but we'll just talk about the snapshot or time slice where just one GCC is running. GCC is an executable. You're first C complier was probably written in C – not written in C, was written in Assembly, but then it kept bootstrapping on the original compiler to build up more and more sophisticated compliers. So the C complier was written in C, I'm sure of it. It also thinks its stack is there and its heap is there and its co-segment where all the assembly code stuff resides right there. I can tell you right now that they're not both all in the same place. They do not share the stack and they do not share the heap and they do not share the code. This virtual picture was in place so that make can just operate thinking it owns all of memory. And it lets the smoke and mirrors that the OS managers, to map these to real addresses and map this to real addresses and map that to real addresses. It just wants to be insulated from that. Maybe it's the case that you have, I don't know, Firefox up and running on one of the Linux boxes, has the same picture. And then you have some other application, like Clock or whatever you have, up there and it has the same exact picture. Those are four virtual address bases that all seem to be active at the same time. I'm gonna call this process one, I'm gonna call this process two, call this process three, and I'll call this process four.

When I draw these little bands of segments right here, these segments is what they're called, they're all assuming that they have as much room to stretch out to make the stack as big as necessary, the heap as big as necessary, to meet the demands of the program. But on a single processor machine, there is only one address base. This is the real deal right there. That's physical memory. And this has to somehow host all the memory that's meaning to GCC and Make and Clock and Firefox, or I should say the processes that are running according to the code that's stored in those executables. Does that sit well with everybody? Well, it turns out that this and that right there, they may have the same virtual address, but they can't really be the same physical address. The space has to be truly owned, or the values there in virtual space have to be truly owned by this process I've called number one. So what the operating system will do is it will invoke what's called the memory management unit to basically build a table of – let's say that this is address 4,000. It'll actually build a table of process and something related to an address. And it'll actually map it to a real address somewhere in memory. Does that make sense? The idea, I think, probably makes sense to people. So that this right here, it thinks it's storing something in address 4,000. Let's say address 600,000 is right there. Any request to manipulate or deal with address 4,000 is somehow translated to a request to deal with the real address at address 600,000. Any type of load or store or access of this right here has to somehow be proxied or managed by this daemon process behind the scenes, this thing that just runs in the background all the time, to actually map virtual addresses to physical addresses. And it knows that this address 4,000 is the one that process one owns, so it just has this little map of information – I've drawn it has a map of pairs to physical addresses – so it knows exactly where to look on behalf of this process.

It stores them in address 4,000; it updates the four bytes that resides at address 600,000. Now, it doesn't really clip these things down at the four-byte level. Normally, what will happen is it'll allocate things – I don't mean allocate in a malik sense – it'll just associate very large blocks in virtual address space with very equally large blocks in physical address space. So this might be some 1K or 8K block of memory. And if it's ever used, then it's sure to map the same size block or adopt the same size block in physical memory so that address 4,000 through, let's say, address 8,000, would map to whatever this is through whatever this is plus 4,000. So there's some contiguous nature going on between all the things in this virtual address base and what it maps to in physical space. Does that make sense? A lot of work. It's a very difficult thing to implement, certainly, the first time. I've never implemented one of these things. Maybe it's even more difficult than I'm giving the impression it is. But it's doing all this stuff in the background, it's threads, it's all kinds of stuff that makes OS and systems code interesting, but difficult. So we've solved the memory problem. In theory, you can run 40 applications. Usually it's the case that the stack segment is never really that big. It's initially slotted off to be fairly small for most applications. Because unless you're going to call Fibonacci of a billion, it's probably not going to have a call depth greater than 50 or 60. In fact, when you have a stack called f of 50 or 60, it usually means a lot of things are happening. The distance down from main to the subhelper to the 59th power function. I don't want to say that's unusual – maybe 100 is normal, maybe 200 is normal; 2 billion is not. So you know that most activation records are on the order of, let's say that they're 1K. It might be the case that you set aside 64K for the virtual stack or the stack in virtual space. You have two the thirty-second different addresses. 64K is ridiculously small amount of that.

It's like that. So it definitely has space for it. You could be more aggressive about the way you use the heap. You could allocate megs and megs of memory there. It's still going to be a relatively small portion of memory when you're talking about two to the thirty-second different addresses. Makes sense? So the smoke and mirrors that's in place so that every single application can run at the same time and not have its address space, or what it thinks is its address space, being clobbered by other processes. That's managed pretty well by the OS – not pretty well – ostensibly perfectly by this memory management. You can share address spaces across applications, but you have to use advanced unit directives to do that. The part that is not clear, and this is going to become more clear, hopefully, next week and the Monday after it, is how the applications seemingly run at the same time, when there's really only one register set, one processor digesting instructions at a time. I did this the first day of class, but it totally makes sense to do it again here. Forget about Firefox and Clock, let's just deal with Make and GCC, which is what you've really been doing. And think about Make and GCC actually running seemingly sequentially.

You look at them both running – and my hands are sifting over the assembly code instructions. And they're both seemingly running at the same time. That's not what's happening. What really happens if that Make makes a little bit of progress, and then GCC makes a little bit of progress, Make makes a little bit, and this just all happens in this interlay fashion, so fast that you don't see any one lagging over the other one. It's like watching two movies at the same time, where not much is happening. So you can actually follow both movies fairly well, as long as it's clear that both of them are actually running. The argument for two hands scales perfectly well to three hands and five hands and ten hands and 50 hands, as long as the processor has the bandwidth to actually switch between all of the processes fairly quickly. That make sense? Now in a dual processor machine or a four processor machine or a multiple core machines, it can actually really run two processes and four processes at the same time, but you can always run more processes than there are processors on any sophisticated system. If something's running a dishwasher, then it probably can't deal with threading, but if it's actually running some real program, it probably is dealing with a real processor, and the OS can actually dispatch and switch between processes fairly quickly. Makes sense? The reason I bring this up is because that, as a concept, is going to translate, I think, somewhat nicely to the notion of threading. This is multiprocessing. Several processes are seemingly running at the same time, and each process has its own heap and its own stack and its own code segment, and its virtual space.

Slightly different, but certainly related, is the idea that two functions in the same process, one code segment, one heap segment, technically one stack segment. We're curious as to whether or not it's possible for two functions to seemingly run at the same time inside a single process. You know; you've seen this before. Microsoft Office, like you're typing and then while you're typing, all of a sudden in the background, some little paperclip comes up and says, I think you're trying to write a letter. And that happens in the background, and that's because something in the event handlers that actually catch your keystrokes have done enough synthesis of the string to look that it looks like a header of a letter. And so it spawns up this other function that doesn't – it's not really supposed to interfere with your typing, and from a computational standpoint, it doesn't. From an actual mood standpoint, it does, because you actually go down and look at it. But that is an example of a thread that is spawned off in reaction to an event, or something like that. That makes sense?

iTunes, you buy an album of 13 songs – I'm hip; I buy music online – and you check the actual download screen and three songs at a time are actually downloading. Not really. It's really doing – and iTunes is another hand over here. And it's pulling things in in little time slices, but it happens so fast and the time slices are so small compared to what we can detect, that it looks like all three songs are being downloaded simultaneously. There's one process going on there. It's not like it spawns off a different executable called the "paperclip executable." It's just some function to bring up that little widget. When you're downloading music, there's one process that's doing it, it's iTunes. And internally, it has some function related to the downloading of a single song that at any one moment it's allowing to run, seemingly, three times – sorry, three at the same time. Does that make sense to people? So just imagine the scenario where there are two songs downloading at the same time. In that case, they both would be following the same assembly code block. They have the same recipe for downloading a song, for the most part. In that case, the stack segment itself could be subdivided into smaller substacks, where the stack frame for song one could be right there, and the stack frame for the downloading of song two is in this completely unrelated space – not unrelated. It's in the same segment, but far enough away that you're not going to have one stack run over the top of another one. Do you understand what I mean? So when the process has the processor, it also subdivided its time to switch back and forth between the two threads – that's what you call these things. And so basically, it goes you have time, you have time, you have time, hope you're downloading songs. And it keeps doing that until one or both of them ends. Does that make sense? It shares the heap. So they all share the same call to malik and they all draw their memory from the same memory pool.

There's only one copy of the code. You only need one copy of the code. It's read only. It's fine for this stack and this stack to both be guided by the same set of assembly code instructions, as long as each one has some kind of thread identifier. If the word thread is bothering you, just think take the phrase "thread of discussion" or "discussion thread" and just translate it to "function thread." Does that make sense? You may ask what types of scenarios actually require threading and which ones really don't require threading. Let me just go over this very simple example where you really would expect threading to be in place to model the real world situation. I will revisit this example on Monday. I have this really simple program, int main – and there's no threading whatsoever. I have this four loop, int num agents. When I call this num agents, I'm actually thinking about ticket agents who answer the telephone at United or some other airline that still hasn't declared bankruptcy yet. Num agents is equal to ten, and let's assume that the job of this program is to simulate the sale of 100 tickets for the only flight that happens to be flying anymore. You might do it this way. You might intrinsically hard code the 100 in there by calling some function "sell tickets," where you pass in i and you pass in ten.

And I don't need to tell you what sell tickets – I'll write code for sell tickets in a second. But the idea here is that we have to sell 100 tickets. This is the number of agents right here. That number is the same as there. In fact, let's say that there are 150 seats on the flight, so we don't have to mix ten's. The idea of "sell tickets" is that it's supposed to be in place to simulate the requirement that some ticket agent actually sells 15 tickets before his or her job is done. As long as these run in sequence, eventually you'll get to the point where you actually sell 150 tickets in this really rude but simple simulation. The problem here is that in the real world, it's just fine for all ten of these agents to be answering telephones simultaneously. It's not – I don't want to start introducing thread functions yet, but I just want to leave you with the idea that we're going to be able to get that function right there to run in ten different threads. In other words, I'm going to spawn off ten different threads. All of them are going to be following the same exact recipe, where each one has to sell 15 tickets, and when a particular thread exists, we just know because of the way we code it up that 15 tickets have been sold. I'm not going to write code for this yet, but this is, I think, a fairly good analogy. Imagine a horserace or a dog race. Sequentially, if you wanted all ten dogs to get to the finish line, you could just let one go, and when it gets to the finish line, let the next one go. And just do it that way, and take 15 times as long or ten times as long as you really need to. Or you can line them all up in ten gates and lift the gates at once. And some will be faster than the other ones, but eventually, they're all going to get across the finish line. Like you're basically pipelining and taking advantage of the fact that things can happen more or less at the same time. Now the difference is that they don't run like this every time, and they're not respecting time slices.

They actually are really independent agents. But we're going to try and simulate that idea as much as possible with this example when we introduce the thread library next time. So this is a great place to stop because I’m at the end of a segment in the course, and to try and introduce threading in five minutes would be pretty difficult. I will see you on Monday and we’ll talk more about threading then.

[End of Audio]

Duration: 44 minutes

ProgrammingParadigms-Lecture15

Instructor (Jerry Cain):Here we go. He's obviously ready. Welcome. I have three handouts for you today. I put them up on the board because they're not in sequence. I went back and used the No. 4 Assignment 5 that I gave out last week, a week ago today. So people who are watching on TV make sure you go back and get the solution to handout 19. I think that's what it is. Let me just check, yes, it is indeed. So there's all of these handouts in the last week that have corresponding solution sets, so make sure that you actually download the solutions as well so you can compare your answers to mine and make sure they're in sync with one another. You know you have a midterm on Wednesday evening, it's 7:00 to 10:00 in Hewitt 200. It's a huge room over there; plenty of space as well as the other room, so it'll be a great place to hang out for three hours. I want to be clear that I am certainly covering co-generation; you've seen that on the sample exams that I gave out last week. I am not gonna include co-generation for C++ features, no true references and no object orientation, no methods. So pointers, that's fine, anything related to asterisk, that's real C and I've emphasized that in the early part of the co-generation, but references and methods, the co-generation for that it's not testable in the midterm. You will certainly see it on the final. I always know exactly what type of questions I put on the final for that stuff, but you will not see it this Wednesday. Okay? I also promise to not have any preprocessor or linker or compiler stuff. I went through that kind of as a transition from C figuring out how to build executables from the C language, but I'm not gonna test that material because I have tested it in the past and it just never goes well because it's so esoteric and we don't have any kind of real problems to exercise the materials. So people just didn't do well, so I just stopped testing it. When I left you last time, I had written this partially simple program that was supposed to model the selling of a 150 airline tickets on a single flight. So let me repeat that and point out why it's problematic and how we're gonna move away from it.

I went ahead and did something like this. I wrote it a little bit differently last time, but I'll write it like this, num tickets is equal to 150. All I want to do, in this brute force four loop, I'll write agent is equal to one agent less than or equal to num agents, agent ++ – I went ahead and I called this function, called sell tickets, and I'm gonna frame in terms of the agent ID num tickets dib, not agents, so that it's planarized in terms of these two values right here, and [inaudible] each ticket agent knows that he or she has to sell that many tickets as part of his or her function call and that's it. That's return zero to satisfy the compiler. Okay. We're not gonna allow anything to go wrong in this simple program. The implementation of sell tickets – it's not gonna be rocket science. I'm gonna write it a little bit differently than I would traditionally write because I'm paying forward to the way we're gonna change the example to in a second. Void sell tickets, int agent ID, int num ticks to sell, and even though it's a little weird, let me not use a four loop, let me use a wild loop. Wild loop is the case that num tickets to sell is greater than zero – go ahead and do print F agent percent V sells a ticket. Agent number and then does num tickets to sell minus minus, finally, my arms tired, but I'm gonna keep on writing, print F agent percent D all done. Agent num – this arms gonna be bigger than the other one by the end of lecture. Okay. There we go. From a code standpoint, it's moronically simple. I'm not trying to revisit four loops and wild loops. What I'm more interested in doing is figuring out why this is as simulation is really not all that good in the sense that it's not really modeling what would truly happen.

The way this is set up, and I'm speaking as this is new material, but it's not, it's clearly gonna be sequential and ticket agent one is gonna sell all of his or her 15 tickets before anything happens with ticket agent two. So you know that the print out of this would have a 160 lines, 16 lines per ticket agent – okay, I'm sorry, yeah, 16 lines per ticket agent, but they would be sorted all ticket agent number one followed by an all done comment, that's the 16th. Okay. Does that make sense to people? Okay. I'm sorry, 165 because there's 15 agents going on here – no, I'm sorry, there's 10 agents so that's 160. I'm just confused, but everything's gonna be all about agent number one before it's agent number two, before it's agent number three, etcetera. I really don't like that. Okay? This is a fairly compelling example where if we're really trying to model the simulation of an actual airline ticketing room that you want to see all of these ticket agents running simultaneously and working, not competing, but working collaboratively to sell all 150 tickets at the same time. Now, what I'm gonna do is I'm gonna repeat the four loop, int agent, agent, less than or equal to the agents, agent ++. Here is how you set up the actual dogs at the racetrack. What I want to do is I want to create a name that's unique to a particular ticket agent. I'm gonna do that by declaring this in place buffer – you haven't seen this function, it's not the emphasis, but I might as well show it to you cause it's in the handout. I'm gonna print F, not to the console, but to a character buffer, there's a function called S print off to do that rather than actually echoing the characters to the screen, I echo the characters in place to a character array and make sure it turns out to be a C string.

The place where they function as that console of sorts begins at the name address, okay, and as long as I don't print more than 32 characters I won't overrun the boundaries of this thing. This is the structure of what gets printed and then I will fill in agent. So for all intensive purposes, on the zero generation or the first generation of this thing, after that S print off call is made, name contains – it goes from garbage to the C string, agent one thread. The reason I do that is because I want to call this function called thread new and the name actually serves as the name of the thread and it's also helpful for debugging purposes should you be passing a true to that thread package. And then you go ahead and you pass the address of the function that you'd like to execute in a single thread of execution. Okay. This right here is just an arbitrary function pointer. All of the arguments have to be four bi arguments with just the constrain of the system so you can pass ints and floats and pointers, but you can't throw in struts or characters or shorts, it'll confuse the system. You have to tell it how many arguments are expected of this particular function. We're gonna have scenarios where the thread functions don't need to take any arguments. We're gonna have a scenario like we have right here where sell tickets takes two of them. Beyond the two, you pass in the numbers that are of interest, so agent and num tickets divided buy num agents. Now, this does not actually prompt sell tickets to start executing. All it does is it lines it up at a gate.

Student:[Inaudible]

Instructor (Jerry Cain):The prototype of thread new just requires a thread name right here. We have to do it because it's [inaudible] but that's a lame answer. You really do want something available to you to identify a particular thread, that for instance, is failing you during the debugging process, and if you have 15 of these things, does that make sense? I'm sorry, you have 10 of these things and then one of them is falling or one of them is never exiting, which is actually a common thing we'll see in multi-threading. You want to be able to know which of the threads is not exiting so you can go and look at the particular implementation of that function. Okay.

Okay. So, procedurally, what happens up front, as we say, we're using threads and then you lay down gates one through 10, all of these things that are gonna follow, the sell tickets recipe, that's what they need to follow in order to run from the starting gate to the finish line and then this basically sounds the bell, fires the gun. Okay. This is a function block until all of these threads actually finish and run a completion and then when all 10 threads are finished, this returns and it passes on to what will be the end of the entire function.

Student: If I wanted to do the same process somewhere else, is this the int package and run all threads within the scope of the function?

Instructor (Jerry Cain):It actually does not have to be in main, it's just most conveniently put there.

Student:[Inaudible] as well when [inaudible] everything out there?

Instructor (Jerry Cain):What has to happen is that before you call run all threads you have to call the package just exactly once and you have to set up all the threads, whether it's directly in main or through sub functions to set up all of the dogs. Okay. Does that make sense? This actually fires the gun and tells all the threads to start running. It turns out that as threads execute, as part of their implementation, they can themselves call thread new. Does that make sense? The threads themselves can spawn their own child threads, okay, grandchildren threads, whatever you want to you. The ridiculous metaphor I have is that somehow while a dog is in the race it gives birth to three new babies and throws them back to the beginning of the gate, okay, and say's, "Please run," and sometimes there's interesting concurrency issues that can come up with that type of thing, but I'll actually get to that with a more advanced example probably Wednesday or Friday.

Student:[Inaudible]

Instructor (Jerry Cain):This just basically says now we're in thread mode. Okay. Once this has been called all threads that are ever created in the process, even if they're children threads, just aren't executing immediately. Okay. So as far as this is concerned, I want to just invent a function right here. If random chance, 0.1, I want to call a thread sleep function and I'll just pass in and say 1,000. Okay. That thread sleep basically says that as part of execution if a thread is running and it executes the thread sleep function that it pulls itself off the processor for at least, in this case, a second, that number that passes as an argument is expressed in milliseconds. So this means that every time it flips a biased coin and it comes up heads with probability of 10 percent or .1, rather, it'll force it to halt. Now, that's not the only way a thread will halt, but this is a way for you to problematically tell a thread to stop running. Let's forget about thread sleep. I shouldn't have talked about that yet. What actually happens is that when you spawn off two or more threads, even technically one child thread, but two or more is when it's interesting, run all threads establishes something of a heartbeat.

In between the top of every single finger some different function, usually in a round robin fashion, but not necessarily, gets the time slice that exists in between the two finger taps. Does that make sense? So it's, like, agent one, agent two runs, agent three runs, agent four runs, in that round robin manner, okay, and however much progress they happen to make in that time slice is the progress they make. Now, if I don't introduce any randomization it's probably the case that on a real system it would execute the same exact number of assembly code instructions. Okay. Or very close to it so everyone would make exactly the same amount of partial progress with each time slice. Okay. Does that make sense? To make it a little bit more real world, we introduce some [inaudible] process here where things all of a sudden get a little bit random and maybe it's the case that ticket agent one sells two tickets and it's in his or her time slice and then gets pulled off the processor instead of actually being allowed to sell two more tickets. Maybe ticket agent two comes next and sells four tickets; maybe ticket agent three comes next and sells four tickets because this coin flip never comes up heads. Does that make sense to people? Yep.

Student:[Inaudible], which threads [inaudible]?

Instructor (Jerry Cain):Well, the one that's executed. The one that actually calls it. Okay. I mean, it's only called once, but the 10 dogs up there are actually each following this recipe, they each have their own little pointer in to be an assembly code that this compiled to. Okay. And if they happen to jump into this function right here then the thread that is actually is stuck inside that function is pulled off the processor and it's even pulled off what is called the Ready Q and put on this thing called the Blocked Q until this number of milliseconds, in terms of time, elapses. Does that make sense? Yep.

Student:[Inaudible] or run all threads, is there any control over how many [inaudible] cycles each one will get?

Instructor (Jerry Cain):Not in our system. Some very sophisticated thread libraries, I don't want to say very sophisticated, a lot of thread libraries, they don't always give you control over the amount of time that's in a time slice. They do want that to be somewhat regular because they don't want you to have unpredictable results, certainly not during the development process. Even though ours does not, some thread libraries, particularly the one in Java, everybody will be the most familiar with by the end of next year when you take 108, you learn about the thread library there. You can attach priorities. There's only three degrees of priorities in Java. I'm sorry, that's not true – there's 10 levels of priorities in Java where you can assign a priority of one to 10 and then, usually, it's the case that they're all sorted and all the threads with priority 10 execute and run to completion before anything with priority 9 is given time. Okay. But we don't have any of that here. We really just want to think of all threads as equally likely to get the processor for a particular time slice in this round robin manner unless there are other things in place that actually block it from being able to make process.

Okay. So think about this line as basically really not blocking it all or blocking for an arbitrary amount of time. Okay. What I want you to imagine here is what type of print out you might actually get in response to this thread implementation. You might get three print Fs, agent one sells a ticket, it may happen like that. Maybe it sells three, maybe agent two comes next, maybe agent three sells five because that's how much time slice allows, maybe five is actually the most you'd actually see as the number of tickets that sold. But you understand what I'm getting at here. Does that make sense? It would just keep on cycling through all of them. Maybe it is the case after 130 or so lines that, for whatever reason, agent seven all done gets printed and then maybe it's the case that agent eight sells a ticket, agent eight all done and then eventually maybe it's the case, for whatever reason, agent four is the last one to sell a ticket and this is just representative of the type of output you might see from this. Now, there's nothing interesting about this from a simulation standpoint because there's really no compelling reason; from a performance stand point to use threading here except that you're trying to emulate the real world a little bit more. There are situations that you want to go with threading for performance reasons, this isn't one of them. I'm just trying to illustrate the thread package. Yep.

Student: If you didn't do the thread sleeve, would you see agent one sells a ticket, agent one sells a ticket, agent one sells a ticket or is it agent one sells a ticket, agent two sells a ticket?

Instructor (Jerry Cain):You would actually see – the thread library has no notion of what a wild loop is so it's not like it detects it, you jump back and uses that as a signal to pull it off the processor. Let's say that the typical time slice is 100 milliseconds, however many tickets can get sold in a 100 milliseconds is how many would be published. Sometimes it's gonna be partial, maybe you're gonna be halfway through the implementation of a call to print F, right, when it gets pulled off a processor and then when it gets the processor back it continues through the partial execution of print out to complete it, return and decrement the num tickets to sell count. Okay. Does that make sense?

Student: There's no way to [inaudible] two processes that you want to run parallel, but you want them to switch back and forth faster than 100 milliseconds?

Instructor (Jerry Cain):You can. Well, [inaudible] you certainly do not have that. I have to think that there's some thread packages out there that do allow you to control the time slice. It's usually not that high priority. Usually you don't introduce threading into a program to have control over the time slicing, you really just do it to have concurrency in the first place, let the thread manager figure out which thread is gonna make the most progress. In an ideal world, we actually don't want to pull agent one off the processor at all if agents two through 10 are on extremely long phone conversations, and that doesn't happen here, but in a real simulation, you might want agent one to keep on selling tickets if all the others are blocked from something else. Okay. This is just in place to illustrate the thread new and the run all the threads and the initial [inaudible] concept. Okay. Yep.

Student:[Inaudible] doesn't [inaudible], it just sort of pulls that one off the Q and it keeps running?

Instructor (Jerry Cain):That's right. And you have to think that this is actually being called by 10 different threads in this set up. It's only written once. It's, basically, like 10 copies of the same book, but it's not even that. It' actually 10 copies of the same webpage and the webpage itself is hosted on one machine. Okay. Does that make sense? There's one copy of the code, 10 independent threads are following the same recipe. Okay.

Student: Well, and [inaudible] run threads directly after any initial thread package?

Instructor (Jerry Cain):Well, it has to be called – oh, I see what you're saying. In other words, to set this up and maybe call it right there –

Student: Yeah.

Instructor (Jerry Cain):– with the idea that these actually run immediately, I know what you're doing. In our system, it wouldn't work because this is a function blocks until all threads have been completed. So what would happen is you could it an int thread package, you would call one whole thread, but there wouldn't be any, so it would return and then it would go on and spawn these threads that aren't allowed to run because run all threads is actually a return.

Student: Okay.

Instructor (Jerry Cain):Okay. This is just idiomatic. Do this, set up at least one thread to run to make sure that all the work that needs to get done gets done, but in this concurrent manner as opposed to this sequential manner, and then call that just to fire the gun. Okay. Yep.

Student: What happens to all sell tickets as that whole thread contacts [inaudible] thread [inaudible] command?

Instructor (Jerry Cain):It's not inside a thread?

Student: Right.

Instructor (Jerry Cain):It could work. The way that the thread library works, the main thread is also a thread, so as part of as sequential execution, it's really not sequential. It happens to still be in a thread, it just happens to be the main thread as opposed to one of these child threads that's spawned off by what was reachable for main. Now, I have to say I've never tested that because I've never given an assignment or done an example where I exercised the edges of the thread library, I've just kind of gone with the way it was designed to be just so I can make progress. But you could try it when the assignment goes out and see what happens with it. You'll never see a meaningful example from me that actually will realize that. Yep.

Student: You said the thread doesn't know if it's in a wild loop, does it know if it's in an instruction, like, does it know that [inaudible] and then the time went off, is it –

Instructor (Jerry Cain):Right. Right. That's certainly the most interesting part of today's lecture is that right now, you know enough about co-generation, I'm hoping, because you're gonna be tested on it in two days, what we say is that it's not an atomic operation. It looks like it's atomic because it's written in one statement right here, but what happens is that this really corresponds to probably what we would as a local variable, it would be a three-assembly code statement. Does that make sense to people? So it's gonna basically load num tickets to sell and do a registered decrement it by one and flush it back out. This is going to be a complexity that we start to solve in the last 20 minutes of lecture right here. So when it gets swapped off a processor, it could be right before the first instruction of the three that this compiles to. It could actually finish right after the third of the three, or it could be pulled off the processor 33 percent or 66.7 percent of the way through the code block that this compiles to. Does that make sense? You feel every single stall time in sequence. If you use threading, and this is the best example of all of the one that I'm gonna have, I think threading is really important. If you use threading and you spawn off 12 download from BBC server threads, all of them make enough progress, all of them try to open the connect and because that's considered a block at the kernel level, it's pulled off the processor. It's a much more harsh version of thread sleep. But it sleeps for a meaningful reason because it really can make a good progress and the thread realizes that and the thread manager realizes that so it pulls it off a processor while it's waiting for the connection to be established. Does that make sense? Well, imagine that all happening with 12 threads, all of those dead times that are associated with the network connection, they all align and overlap and pipeline in this way that really saves us a lot of time. Does that make sense? Okay. Yep. Go ahead.

Student: The last lecture you mentioned about downloading strings in class; I was thinking if you only have download capacity, you only have so much speed you can download, so what size files can you download in a certain amount of time, how can you download more than that?

Instructor (Jerry Cain):You're actually not. As far as the downloading is concerned, if you're dealing with a unit processor and you're dealing with one processor with one ram and one core, then you're dealing, primarily, only with the ability to index one article at a time as the text comes through. So you're right, you don't save time for the actual pulling of the text and parsing of it, and updating your hash sets, but you really save the time with your network connections, and that's what the huge win is. Okay. Does that make sense? Okay.

So what I want to do here is I want to complicate this problem a little bit, but complicate it in a meaningful way. In a real world simulation, it might be the case that you have two ticket agents, and you have to sell 10 more tickets and if somebody's stuck on the phone because they want to buy the ticket or not, so the other ticket agent should be able to sell all nine or 10 tickets while the other is blocked with some time consuming customer. So what I'd rather do is rather than actual instructing each particular thread to sell a pre-determined number of tickets, I'd rather grant them all access to the same shared integer, the master variable, that stores the number of remaining tickets and do something like this; int agent and int star numb tickets and I'll put a P there. I'll close it off for the moment. I'll change this up here in a second. What I want to do is I want each agent to know what their badge number is, but I also want them to be able to go back to the main function and find the master copy of the number of variables that are remaining. This is basically the equivalent of the one master copy of your checking account balance. Every single ATM machine in the world is supposed to have atomic transactional access to. Does that make sense to people? Okay. Here is the main thread and its stack frame. All 10 other stack frames for the 10 other executing threads all have pointers to that one 150 inside, and that's how they kind of keep dibs on how many tickets there are remaining to sell. Okay. Does that make sense? Now, the problem and this is actually not even the full problem, but I'll simplify the problem to make it seem like it's easily solved, is that I, as ticket agent one, might come through and I might commit to that test and say, "Oh, wow, there is, in fact, one ticket left, that's greater than zero, so I'm gonna commit to selling it."

Make sense? And then boo-hoo, it gets swapped off the processor right after the curly [inaudible], but before anything associated with the num tickets minus minus. Okay. Make sense? So it gets swapped off the processor and thread number two comes in and executes the same test. "Oh, look, there's one ticket left, I'm gonna sell it," and it comes in and it commits to trying to sell it, but it gets swapped off the processor. Same thing for thread three, thread four, it could be this diabolical situation where everybody is really excited to sell the one remaining ticket. They don't go back and recheck the test after they get the processor back, that's not what is probability encoded, so they're all gonna try and decrement this shared global and so one, could, potentially, go down to negative nine. I don't think this is why airlines overbook flights, okay? But you can understand the type of concurrency problem that exists here. They're all depending on the same shared piece of data, and if they're not careful in the way they manage the shared data and if it's partway through the execution and it makes decisions based on information that will become immediately stale if its pulled off the processor then the integrity of the global data can actually be mucked with and be compromised. So, at the very least, we want this all the way through that to, more or less, be executed in full. Okay. So basically what that top bracket and what the bottom bracket does it kind of marks that thing right there, is what's called a critical region. It's, like, once I enter this region, no one else is supposed to be in here while I'm doing surgery on that global variable. Does that make sense? Now, there's nothing in the code that actually says, "Please other threads, don't come in here because I am," there have to be some directives that are put in place to block out other threads. This is the situation where you're really glad that the bathroom door locks because if you're in there, you don't want them to have the privilege of just walking in because they're running in their own little thread. You actually have to have a directive in place, this thing called a lock, I'm gonna frame it as a binary lock, I think for obvious reasons, because you only want one person in the bathroom or in the critical region, right here, at any one moment. Okay.

So what I want to do is I want to talk about the most common concurrency tool that's in place to actually help delineate what is considered to be a critical region. It involves me introducing another variable type. I want to introduce something called a semaphore, and I'm gonna call it lock and I'm gonna set it equal to semaphore new. It takes two arguments; the first one I don't care about, the first one is gonna be some integer. Now, I'm just introducing semaphore like it's a word that you're all familiar with. I know you probably know what semaphore means in a general sense, but in a programming sense what a semaphore really functions as is non-negative integer, at least in our library it's considered to be a non-negative integer, that as a data type has functionality that supports atomic plus plus and atomic minus minus. This, basically, sets this glorified integer equal to one, okay, the minus minus and the plus plus against this lock, comes in the form of two different functions. There's a function called semaphore weight which, in this case, would take the lock variable. There's also another function called semaphore signal, which also takes the semaphore. Now, those are functions that, behind the scenes, emulate minus minus and plus plus, but they just figure out using special hardware or special instructions of the assembly code language to actually take the integer that's wrapped around by the semaphore, in this case, what's initially a one, and provide atomic minus minus. Okay. So, in other words, this right here would be decremented to zero if this were called against it. This would promote it back up to one. The reason that weight is the verb here is because we're gonna generalize a little bit. Think about the semaphore as tracking a resource. In this case, there's exactly one person allowed in the bathroom or there's one person allowed into the critical region, okay, which is why that's a one in the first place and you acquire that resource or you wait for that resource to be available and when you don't need it anymore, you signal it or you somehow release the lock. There's one key that I forgot to make – is that because the semaphore integers in our world are never allowed to go from non-negative to negative, there's a one special scenario that's handled by semaphore weight. If a semaphore weight is passed to semaphore, that at the moment, it analyzes it and is surrounding a zero, it doesn't decrement it to negative one; it's not allowed to do that. That's just the definition of what a semaphore is. If it detects that it's a zero, it actually does what is called block and it blocks on that semaphore.

It actually pulls itself off the processor because it knows that it's obviously waiting, presumably for some other thread to signal that thing before it could ever pass through that semaphore weight [inaudible]. Does that make sense to people? Okay. Basically, if I'm jiggling the door for the bathroom, like we always do at restaurants to wonder whether somebody is really in there or not, okay, you need, before you can really pass in there, you need someone else to release the lock, some other thread or some other agent in the form of a semaphore signal call before you really can go and open that door and then you can look it yourself. What I want to do is I want to do is pass in three arguments to sell tickets. The reason I want to do that is because I want to tell the ticket agent what his or her idea is. I want to pass in the address of the shared resource, but I also want to pass in this thing I called lock. Now, the semaphore type is actually a pointer to an incomplete type. It's not copies, it's actually share some kind of strut behind the scenes that tracks the integer inside of it. And then the prototype of this would be the change to take a semaphore, I'll call lock, and this is the implementation I want to go with. I'm gonna simplify it a little bit. I'm gonna say while true, I'm gonna semaphore weight for the lock. As a thread, I have no business following that pointer and looking at its value and comparing it, using it in any sense, even comparing it to zero, because as I advance through the execution, I can't trust that that comparison is actually meaningful, if at any point during progression, it actually gets swapped off the processor and other threads can go and muck with that shared variable. Does that make sense to people? Okay. So what I want to do is I want to wait on the locked bathroom door and if I happen to be the one that first detects that it's unlocked and I can go in and, in this atomic manner, actually do a decrement. So as I detect that it's been promoted from zero to one, I actually take it from one down to zero and actually pass through this semaphore weight call, then I can do this. Num tickets P is double equal to zero then I want to break, otherwise, I want to do this – I want to print up that I have right here to say that I sold a ticket and then I want to semaphore the signal lock. The one thing I want to do here is that if, as a thread, I acquire the lock and I notice that there are no more tickets to be sold, when I break out I don't want to forever hold the lock on the bathroom. Okay. If you can programmatically unlock the door from afar you're no longer in the critical region, but you still somehow manage to unlock the bathroom door. Now, there's a couple of points I can make about this just to let it rest for you because this is probably where I'm gonna leave things until Wednesday. I initialize the semaphore to one up there.

That basically functions as a true. It basically says that the resource is available to exactly one thread and the first thread to get here actually does manage to, in an atomic way, take the one and do a minus minus on a down to zero because it actually committed to the minus minus, it returns – it executes this. It takes the zero back to a one one. It may come back around and take the one back down to a zero, but it's always, like, lock, unlock, lock, unlock, lock, unlock, lock, and maybe it actually gets swapped off the processor right here. That would normally be dangerous except that it's leaving the semaphore in a state that it surrounds a zero. Okay. So there's some other threads that the processor and it certainly will then they come here and they, basically, are blocked by a zero semaphore. Does that make sense? Okay. Imagine a scenario where I accidentally – and this is actually the type of thing you have to be careful about because it's so easy to type a zero versus a one when you're typing a lot of them. If I do that right there, this creates a situation that you really have to be worried about when you're dealing with concurrency and threads, is that if I accidentally lock the bathroom door before anyone comes to the party, everybody's' gonna be blocked and no one is in a position to actually unlock it. At least not the way I've coded things up right here. Does that make sense? If I make the mistake of putting a zero up there, then every single thread will get this far and they're all gonna be thinking that someone else is gonna be [inaudible] that semaphore, so all 10 of them are pulled off the processor and everybody's just waiting. That isn't the case because of that one little bug that I put up there. Okay. Make sense? If I have the opposite error and I do that right there, from a programmatic standpoint, if it's gonna be two, it might as well be 10. If you're gonna let two people in the bathroom why not let all 10? If you're gonna actually let two people go into the critical region and muck with global data at the same time, then you have the potential for having two threads deal with a shared global variable in a way that they really can't trust each other. Does that make sense to people?

Student: Yeah.

Instructor (Jerry Cain):Okay. So there's that. So the real answer here is that this, in this particular case, should be a one. Now, we will see situations where a zero is the right value. Okay. We will see situations where two or five or eight or 20 or 64 are the right values, but for this one scenario where I'm using a semaphore to basically limit access to what's clearly identified as a critical region, okay, that is the common pattern for using a semaphore. Okay. Question right there?

Student: Do you have two signal locks?

Instructor (Jerry Cain):Two signal locks, oh, this one right here? This is the one that actually is there whenever I actually do do a decrement. Because I can break out of the loop right here – if I break out of the loop, I circumvent this final call right here, but other threads may be blocks on the semaphore right here. All they need to do is to verify, as well, that there are no tickets left, but you still have to allow them to program as they get there so they as threads can also exit. Okay. Does that make sense? Okay. Yep.

Student: Is there something stronger than a semaphore that actually won't let the thread get pulled if you have something time sensitive?

Instructor (Jerry Cain):Actually, just priorities is really it. Even then, it's probably up to the thread manager as to whether or not – what would probably happen is a really sophisticated thread manager might actually know behind the scenes before it even grants the thread or processor, but there's only one thread with that priority. So it might actually have – and I don't know that this is the case and I'm just speaking in terms of implementation details – it might say, "Okay, that's the only one of that high priority, so unless we see a spawn of thread of equal priority or higher priority, we're just gonna let it run until it actually blocks itself," in which case, we don't have any choice. I don't know that many systems do that. It's technically possible to do it. Okay. So we'll have more examples come Wednesday, but I just wanted to make sure that you all got this. Have a good night.

[End of Audio]

Duration: 53 Minutes

ProgrammingParadigms-Lecture16

Instructor (Jerry Cain):Hello, welcome. I don't have any handouts for you today. You have plenty of handouts from Monday that we still have to spend the next few lectures on. You're not getting an assignment today, so you have this grace period from tonight at 10:00 p.m. until Friday, where you have no responsibility for 107 whatsoever. Remember, the exam is this evening at 7:00. It's in Hewlett 200, which is this huge auditorium in the building across the – beyond the fountain from gates. I'm going to send an email out after lecture, but just in case SUPD students are watching this before the TV exam tonight, I'm planning on posting the exam as a handout at 7:01 p.m. tonight, and then remote students just download it, self-administer, call in if they have questions and then fax it in when they're done. They don't need a proctor. I don't need any of that business. I just assume people are well suited to just sit by themselves and take an exam without somebody of authority hanging over their shoulder. And then, SUPD students actually have the option to take it tomorrow morning as well, and I actually prefer that SUPD students take it, because if there's a disaster during the exam tonight, people in the room can be dealt with immediately, where as it's very difficult to probably get that information outward. So, I actually prefer SUPD students to take it tomorrow and then fax it in sometime before 5:00 tomorrow so we can grade them.

Okay, I'm going to try to get the graded exams back to you and available by Sunday evening. I can't promise that. Okay, I actually have not dealt with a class this large in a long time, so we're dealing with – I know it looks like it's this cozy little family here, but it's not. It's actually 230 some people and it's been a while since I have had to manage a grading effort that involved that many people. It's also complicated by the fact that I am out of town this weekend, so my CAs are grading it and it might be difficult for them to get you the exams back by Sunday evening, but we'll do our best to make sure that that happens, okay? When I left you last time, I had focused specifically on the first multi-threaded example where we had to introduce this notion of a semaphore in order to control access to what we called a critical region. So, if you remember last time, the threaded function, the one it's the recipe that ten different dogs follow while they're trying to get their work done. It looked like this. Sell tickets, token INT, agent, it took an INT star called non-tickets and non-tickets P and then it also took this semaphore, I call lock. Just to review, since this was kind of a fleeting comment in the last ten minutes of Monday's lecture, the semaphore – it is more less like a – basically let's say a synchronized counter variable that is always greater than or equal to zero, okay? And so if I construct a semaphore around the number one, and I levy a semaphore weight call against the semaphore, this as a function figures out how to atomically reduce this one to a zero. Okay, and so this is basically equivalent to the minus minus but it does the minus minus in such a way that it actually fully commits to the demotion of the number to one that's one lower than it, okay? Semaphore signal on the same exact semaphore would actually bring this back up to a one, okay? If this were followed by a semaphore weight call followed by another semaphore weight call, then something more interesting happens, where this one right here decrements the one down to a zero. This one right here would have a very hard time.

Because semaphores at least in our library – this isn't the case in all systems, but our semaphores are not allowed to go negative. So, when you do a semaphore weight against a zero variable, then this thread actually says oh, I can't decrement that, at least not now. I need somebody else in another thread to actually plus plus this so that I can actually pass through a minus minus without making the number negative. Does that make sense to people? Okay, so programmatically, the implementation of semaphore weight is in touch with the thread library and so it actually when it's attached to a zero behind the scenes, it immediately says ok, I can't make any progress right now. It pulls itself off the processor. It records itself as something that's called blocked and it puts it in this cue of threads are not allowed to make progress until some other thread signals a semaphore they're waiting on, okay? Does that sit well with everybody? Okay, this is not constrained to go between one and zero. It can – this can be set to either be zero or one or five or ten. The only example we've seen so far is where the semaphore that's coming in is initialized to surround the one because we really want it to function not so much as one as we want it to function as a true, and it's basically a light switch that goes on and off, on and off, and on and off, and it's used. And, we use semaphore weight and semaphore signal against that semaphore to protect access to the non-tickets variable that we have been addressed to.

So, in a nutshell, while it's the case that true is true, I want this right here to be marked as a critical region. What that means is that I want to be able to do surgery on that non-tickets variable, without anyone else bothering it, okay? And the way you do that is to do a semaphore. I'll spell it out here. Semaphore weight on the lock, you do the check to see whether or not non-tickets of P is equal to zero and if so, you don't do the surgery. Okay, on the end, somebody else has done it a hundred times already and there is no reason to do it again. Otherwise, you want to go through and do this, that's true surgery on what functions as a global variable, at least from the perspective of all the threads running this and then you want to release the lock. You might do some printing here, okay? You might sleep for a little bit. Down here there was an extra semaphore signal call against the lock to accommodate the scenario where the person who breaks out of the wild loop does so after securing the lock, so they actually have to release the lock kind of as they go outside the bathroom window, as the analogy I used on Monday, okay?

This is considered to be – I'm sorry, this right here is considered to be a critical region. It's supposed to be something that when they're inside there, they cannot have any other threads during their time slices mucking around with this type of stuff, okay? As an arbitrary thread actually gets here, given that this thing was initialized to one, and because every single semaphore weight call is balanced by a semaphore signal, it's going to toggle up and down between zero and one. When a thread gets here, there is one of two scenarios. It's staring at a one and so it actually successfully does the minus minus and is allowed to pass in here and do the work or the thread blocks on this. You may say, well, how would that happen? A thread could potentially block on this if there's a zero, if another thread saw a one here decremented to a zero, made partial progress through this but the time slice ended before it got to the signal call. Does that make sense to people? Yes? No? Okay, just because a thread owns a lock doesn't mean that it can't be pulled off the processor. It might acquire the lock; get two-thirds of the way through this final instruction here, okay? And then be pulled off the processor so that other threads can actually say, oh maybe I can make some progress, but if they get this far, they're still seeing a zero because of the thread that owns the lock hasn't released it yet.

Okay, so as other threads hit this semaphore weight call and it surrounds a zero, they are pulled off the processor. That is kind of what you want. If they can't do any meaningful work, you want the thread manager to say you can't do any meaningful work. I'm not going to let you even use your full time slice and eventually it'll get back to the only thread that can do work, which in particular would be the one that owns the lock right here, okay? Does that make sense? Okay, so there's that right there. Typically, try to keep the critical regions as small as possible. If you're going to lock down access to code, you don't want to make it arbitrarily long. That's basically like saying I want to do all the work in the world, so I'm just going to acquire a lock and I'm going to run this triple four loop. Okay, you only do that if you have to because it's a critical region. This print up in particular, if it's just logging information, it might not be imperative that you actually print to the console while you hold the lock, so you could release the lock and let other people make some progress, and then without holding the lock, just go ahead and print to this screen, okay? Does that make sense? Okay, there are a couple of other things. Somebody asked a very good question at the end of a lecture on Monday and I think I want to go over it. Some people were concerned with the case where non-tickets paid minus minus takes a one down to a zero, I don't mean the lock, I mean the actual number of tickets and they thought that was the one problem we were worrying about.

The answer is that's not the case and I can actually tell you a little bit more about what happens as threads get swapped off the processor and where all of their data gets stored, and show you that if the number of tickets is originally 100, there is as much of a race condition without the semaphore weight and semaphore signal calls in the minus minus bringing the 100 down to a 99, as there is in bringing a one down to a zero. So, this is what would happen and this is going to be the most important line to concern ourselves with. Forget about the fact that there are ten or 15 ticket agents. Just think about the scenario where there's two. Okay, it is subdivided and this is the main stack frame. Okay, it's the thing that sets up the two threads and caldronal threads. Let's say that this is ticket agent one and this is ticket agent two. Okay, these little like tornadoes are actually stack frames, okay, that each of the threads has. So, each of these things right here have their own activation records for their own call to sell tickets, okay? Declare someone a name is the number of tickets variable that will set equal to a hundred. That make sense? Each of these stack frames stores a pointer to that one hundred, okay? Imagine the scenario where the semaphore weight and semaphore signal are not there. This is how the two ticket agents could each sell one ticket even though it's only globally reported as a single sale, as opposed to the two tickets that were really sold. Coming down from here, la la la la, you see that it's not equal to zero, so you go in and try to sell a ticket. You know that this type of instruction actually expands to quite a number of assembly code instructions. Okay, does that make sense to people? So, in a local register set, this R1 may be set to .2, that right there, or two may be set to 100, because you do R2 is equal to that of R1.

Okay, then maybe you go in and you do a minus minus on R2, and bring it down to a 99, but now you're swapped off the processor. So, you've actually committed to the sale of a ticket, right? But, you're swapped off the processor. What happens is that the entire register state, all 32 registers, including the PC and the SP and the RV registers, if this is the binary state of all those registers, it's actually copied to the bottom of the thread that's being swapped out, little stack frame. Okay, and is that image that I just drew, that is used to restore the register set for that thread when it gets the processor back, okay? Embedded inside this image is the 99 that's going to be used to flush back to this space right here. Does that make sense? Okay, but this has just left the processor and so this thing does exactly the same thing with the register set as if it owns it. It sets R1 equal to that right there. The 100 still resides there because we didn't successfully flush back. We didn't get to the point where we actually update it to be the decremented value. So, R2 gets set equal to 100, gets set equal to a 99. This one is prepared to flush with a 99 back to the global space when this thing gets the processor back; it's going to flush a 99 back to the same space. So, this 99 is designed to override a 100. So, is this one, but unless you have semaphore weight and semaphore signal in place the way we do right here, one of the 99s is going to override a 100 and one of the 99s is going to overwrite the other thread's 99. Does that make sense to people the way I'm saying that?

Yes? No? You gotta nod your head, okay. If you put that there, and you put that there to balance it, and basically unlock the door, then only one thread is allowed to go through and actually pull the global value into the local register, decrement it locally, and then flush it back out to the global integer, the thing that functions as a global integer, because everybody has a pointer to the same integer. Before any of the thread is allowed to do any part of that, okay, so that is what I mean when I say that this is more or less committed to atomically, okay? Now, that is the overarching principle that is in place when you have threads, and in particular, you have threads accessing shared information, okay? This is the programmatic equivalent of the two Wells Fargo ATM machines where me and my best friend try to take out the last remaining $100.00 in my account at the same time, thinking we're going to get $200.00. Okay, make sense? Okay, if I were to initialize this semaphore to zero, then I would actually block all threads from entering this critical region right here. Okay, so I'd get deadlock. If I initialize the semaphore not to one but to two, that's as bad in principle as initializing it to ten, because you don't want any more than one thread in this region at any one time, okay. Does that sit well with everybody? Okay, good.

Okay, so let me move on and give you another example using threads and a different way to use semaphores. The handout actually uses global variables more than I like to, but this next example, I am going to use globals just so the code matches up a little bit more cleanly with the handout version. I actually like it better if you declare all of the shared variables in main and pass addresses to them, to the threads, because at least everything has a scope to it, whereas globals, it's a free for all and for two and a half quarters, we have been saying globals are awful. Oh, except for when it's convenient, okay, and I don't like that. But, this one next one, I want to frame it in terms of globals. I'm trying to model right now the Internet, where in all the world there's only server that serves up all the web pages and you have the only other computer with the only browser in the world.

Okay, I know you know enough about the HTML server process. You may not know all the mechanics at the low level, but fundamentally, you know that you request a web page from another server. It serves up text in the form of HTML normally. It could be XML, but normally it's HTML, and as the HTML comes over, it does a partial and eventually a full rendering of the HTML in your browser page. That make sense?

You know and you felt this before where a page is loaded like 70 percent but it's not quite done yet, you see the progress bar at the bottom, the bottom right, where it's like three-fourths of the way through and you know there's more to come. That's usually because the server has only delivered 75 percent of the content and so this thing has to block in much the same way that the threads up there block, this has to block and stop its rendering process until it gets more data from the server. Does that make sense? So, just use that as a guiding principle for this example. I'm going to insult all the Internet and I'm going to reduce it to a character buffer of size eight, okay? And what I want to do is I want to write a program that simulates the writing and the reading process and I'm just going to reduce the server to something that has to populate that buffer as a ring buffer. In other words, it's going to write 40 bytes of information, but it's going to cycle through the same array five to five times and I'm going to write another thread that consumes the information by cycling through the same array five times and digesting all the characters that are written there. Does that make sense to people? Okay, so the main function – I don't care about those. I have to emit all threads, I'm sorry, that's not right. It's the hybrid of two functions. I want to emit thread package of [inaudible] meaning I don't care about the debug information, and then I want to do this. I want to call thread new twice. I'm going to give them both names. This one is going to be called Writer. This one's going to be called Reader. Okay, I'm going to call the function writer and I'm going to call the function reader. I only have one instance of each one and neither one of them takes any arguments. Okay, it doesn't need to take arguments if you use global variables. Then, I do this. Run all threads. This is somewhat pathetically, but on the well intentioned – it is trying to emulate the fact that the server and the client as computers are both running at the same time.

Okay, programmatically I want the writer thread to cycle through and write 40 characters to the Internet. Okay, I want this reader thread to consume the 40 characters that are written in the Internet. Okay, this is what the writer function looks like at the moment. Four INT I is equal to zero, I less than 40, I plus plus, da da da, what I want to do is with these iteration, I want to call some function. I'll assume it's thread safe. Prepare random car and then I want to write to buffer of IMOG eight, whatever, let's give that variable a name, C. Whatever C happened to become bound to and so as an isolation function, I think you can look at this and understand that it's going to write down random characters in this loop over the buffer five times. Okay, I write this with hopes that it writes data down before the reader consumes it but it doesn't go so far that it clobbers data that has yet to be read. Does that make sense to people? Okay, let me write the reader, which has the same exact structure: I less than 40, I plus plus, there you have that. What I want to do is I want to basically do this and then I want to basically like, you know, process car, which I don't care about the details of what process does. This is the consumption line. This is the thing that takes a meaningful piece of data in shared space and brings it into local space, so it kind of owns it. It can do whatever it wants to with it. Okay, let me draw the Internet. That was easy. There you have it. Now, you know, without concurrency, you know exactly how you want writer and reader to behave, so that everything is preserved and the data integrity is respected, and that the reader processes all the character data in the order that the writer writes it.

So, think about the scenario where the writer gets to run first and its first several time slices, it writes those three characters down. Okay, and internally it has a variable of I that is associated with that index, so that's where it'll carry on next time. Okay, but it writes those three variables down. And, then the reader gets a time slice and for whatever reason, process character is a little bit more time consuming. It actually has to like open a file or a network connection or whatever it has to do, just pretend that it actually is slower – its [inaudible] is slower, so it only really consumes that A. It doesn't really remove the A, but it just consumes it so it doesn't matter what's there anymore. Okay, so this is where the writer will pick up and this is where the reader gets swapped off, okay? I think it's pretty clear that if the writer is able to make more progress per time slice than the reader, then there's the danger that this might happen. And that on the very next iteration, it gets far enough in its time slice that it overwrites data that the reader has yet to deal with. Does that make sense? Okay, you can't have that obviously. Now, clearly, I'm simplifying things here, but the idea that someone is providing content and someone else is digesting it, that's not an unfamiliar one with large systems. It's also in theory possible. Just because I spawn off and set up writer to be the first thing that runs and this the second thing that runs, it might be the case that the reader gets the processor first. In which case, it will be digesting information that has never even been written or created. Does that make sense?

So, what I want to do. I want to create Internet so I can put some more global variables here. I have to make sure that the writer never gets so far ahead that it's clobbering data that has yet to be consumed. I have to also make sure that the reader never gets – never catches up or passes the writer and consumer information programmatically that isn't really there. Does that make sense? Okay, so what I could do is I could introduce two global integers and have semaphores that walk them down, but I'm actually going to use semaphores a little bit differently. I'm going to declare two semaphores here. I'm going to call one empty buffers and I'm going to call one full buffers. And, I'm going to let them actually manage integers that are always, almost always, but we're going to pretend always, are always in sync with the number of slots that can be written to and the number of slots that can be read from. Okay, I also want to enforce that the writer is also just a little bit ahead of the reader in terms of thread progress and that the reader can get and catch up to the writer, but it can't pass them, and that the writer can't get so far ahead of the reader that he actually is more than a cycle ahead of him. That make sense?

Okay, so what I want to do, I'm not going to do the semaphore new column. I'm just going to say that this is going to be initialized to eight as a semaphore. That is not the syntax floor but that's conceptually what I want to happen. Okay, I want to mention that up front that there are absolutely no full buffers whatsoever. Okay, make sense? I'm going to change this function right here to do this. Now, this is a slightly different pattern with the semaphores, but I think it's really fun. Before I go ahead and write to this buffer, I better make sure that I'm allowed to do that, okay? What I'm going to do is I'm going to semaphore weight on empty buffers. Now, initially, empty buffers is equal to eight, which is consistent with the fact that we don't care if the writer makes a lot of initial progress. Okay, but if for whatever reason and the writer makes so much more progress than the reader that he gets really far ahead, this eight will have been demoted to a seven, to a six, to a four, to a two, to a one, and it really is just about the clobbered data that has yet to be consumed. It will be waiting on something that will have been demoted so many times that it's actually zero, okay? So, it will be a victim of its own aggression and it will be blocked out and be pulled off the processor, so that the reader can actually do some work. Okay, the balance here is a semaphore signal call but it's not against the same semaphore. After you write something down, you want to communicate to the reader that there is even one more piece of information that it's allowed to consume. So, I'm going to wait for something to be empty. I'm going to change from empty to full and I'm going to signal the full buffer semaphore, okay? The pattern over here is somewhat symmetric.

Let me rewrite it, is that I want to do the same thing, semaphore weight, but I want to wait for there to be a full buffer. When I know that there's at least one and I pass it that semaphore weight call, I can consume the character that is in global space and pull it down and then after I bring it to local space, I can immediately tell the writer that it's okay to write there if they're waiting, and then process car pass the C. Okay, there's that. Whoops, so it's like each thread has a little buzzer. Each of them are twittering each other as far as when they're allowed to proceed to read or write information. Does that make sense? This right here is sending a little buzzer that allows that to execute and return with much more likelihood. This right here is really communicating to the thread at that point and promoting full buffers so that the writer can actually write down more data, if it was previously blocked. Does that make sense? Okay, so think about what happens now. Empty buffers is eight, full buffers is zero. That means the writer has all of this free space to write to. It's going to have a very easy time passing through the semaphore weight call initially. Whole buffers is zero. The reader thread is bumming because the very first thing it has to do is semaphore weight on something that is set to zero. So, imagine the scenario where the reader actually gets the processor first. It's going to execute this much. It's going to declare I, it's going to consider to zero. It's going to pass the test. It's going to come down here and it's going to be immediately blocked from this line right here because it's going to be waiting on something that is in fact zero.

Okay, so the reader thread is actually being blocked right up front just like we want it to be. Okay, the other scenario is that the writer thread really fast and very efficient, it actually cycles through this thing eight times and then it hits a wall. Okay, so pair the character out before it actually went to bother on waiting on the lock, but – and it blocks here, it's because it's been a processor hog and it's actually done a lot of work whereas the reader hasn't really been able to do much at all. Okay, or at least comparatively. That make sense, people? Okay, so the ticket agents example where it uses a semaphore weight and a balance semaphore signal on exactly the same semaphore, and it brackets this thing called a critical region, that semaphore pattern or that semaphore is being used as a binary lock. Okay, binary meaning it's toggling between zero and one, true and false;

however we want to think about it. That's not the pattern that's being used here. We certainly have thread communication. We use the semaphore for rudimentary thread communication, okay, but right here what's happening is we're actually using these as basically two telephone calls. Okay, between the two threads, okay, this one calls this one whenever it can make more progress. This one calls this one whenever the writer can make more progress.

That is a pattern; it's what called a rendezvous pattern. Like I'm syncing up with you, that kind of thing, okay? There are more complicated examples of this. This is what's called binary rendezvous which really just – one thread to one thread communication. This basically says as this type of semaphore weight call means I cannot make any progress until some other thread makes some required amount of progress in order for me to move forward. This thing does the same thing on behalf of this semaphore. This says that I have to wait for some other thread that makes enough progress in order for me to pass, okay, or else the work I will be doing will be meaningless, okay? Make sense? Okay, so what I want to do is I want to just experiment. What happens if I make that a four? It doesn't change the correctness of the code or it depends on how you define correctness, but you will not get deadlock, okay? And you will not have any integrity data issues, all you're constraining is that the writer and reader stay within more of a delta of one another than they would have been able to otherwise. When it was an eight, it allowed some more degrees of freedom. It allowed the writer to go much further ahead if that's just the way thread scheduling worked out. When I made it a four, it just means that the writer can be no more than half of an Internet ahead of the reader, okay? Does that make sense? If I do that right there, I'm really pushing the limits of what's useful from a threading standpoint. If I'm going to do that, I also will just actually write the reading and writing in the same function and have it alternate between read and write, but if I really let these two threads run with those two initial values, all that's going to need – this is my W finger, this is my read finger. It just means it's going to run like this. Okay, does that make sense to people? And really if it tries to like run forward two slots, it'll be blocked by a semaphore weight call.

Okay? If I do this and I have a different form of deadlock, but deadlock is deadlock. I have a reader saying I can't do anything because I have no place to read from. The writer says well, I can't do anything because I have no place to write to. Okay, so you would have deadlock. You look at that and you say I would never do that, yes, you would. You just have, like when you're writing down all of the semaphore values, maybe you have like 20 semaphores in a real program, it's very easy for you to cut and paste a zero in place where you really wanted a one or a four or an eight, okay? So, if you have deadlock and you've never had that before, maybe you have because you've been in some wild true loop, but that's not the same thing. You really are making progress, you just don't see it. With threads, if you have deadlock, everything seemingly stops. You get nothing published to the console at all. It doesn't return. You don't get your command line prompt back, so things just expand and then you go okay, that probably means that two threads are waiting on each other.

Okay, or that nobody released a lock or something like that. Okay, if I do this, just think about whether that's damaging or not. You may think that initially [inaudible] should be more than full buffers. Let me do this. You could say well, I just want to kind of constrain full buffers plus empty buffers to always be eight. Okay? But if you do that, that actually allows the reader thread to get one hop ahead of the writer. Okay, so that's a kind of contrived example, but nonetheless that's exactly what it will be permitted to do. It doesn't mean it would actually happen, but it means programmatically it's possible. Another scenario is when I get this one right, but I do something like that. Okay, you may think that you're limiting things because you have semaphores in both directions but that thing has to be between one and eight for it to be programmatically correct. To put a 16 means that the writer is allowed to make two loops and take two tracks or two loops on the reader thread and that's not allowed. It's supposed to be at most eight slots ahead of, not 16 slots ahead of the reader, okay? Now, I had one and four and eight there before, my argument is that it should be the eight. Okay, if you have multiple options as to what you can initialize your semaphores to be, you always error on the side – although error is not the right word – you always kind of move toward the decision that grants the thread manager the most flexibility as to how he schedules threads, okay? And it also improves the likelihood that every single thread will be able to use all of its time slice. Okay, to the extent that you artificially constrain the threads, if you were to make that eight a one again and you get this again, it probably means that each thread is being hiccupped and pulled off a thread prematurely. I'm pulled off the processor prematurely. Does that make sense? Okay, and so you usually try to maximize throughput and you choose your semaphore values accordingly. Okay, does that sit well with everybody? Yeah?

Student:[Inaudible] semaphore weight?

Instructor (Jerry Cain):Which one is this? This is on the semaphore you mean?

Student: So, you called your semaphores, so now [inaudible] semaphore weight, so –

Instructor (Jerry Cain):That's actually not. You mean you call semaphore here before you wrap around and wait on empty buffers?

Student:[Inaudible]

Instructor (Jerry Cain):Well, this one never waits on full buffers. That one does.

Student:[Inaudible] signal before you call semaphore weight?

Instructor (Jerry Cain):Which semaphore weight are you talking about? Oh, I see what you're saying. In other words, if the reader doesn't agree with the processor, what happens here – is it the writer just happens to go first? It brings an eight down to a seven and it promotes a zero up to a one. But that's okay because it really is one slot ahead of where the reader is. The reader hasn't even started yet.

Okay, and so if this makes, let's say four full iterations and it brings empty buffers down up to four and full buffers down to four, that's fine, because if it gets swapped with the processor here, this thing just discovers a world that it's born into – it says wow, there are four characters I can read right away. Okay, and so it doesn't matter that it hasn't blocked – it hasn't called weight yet.

If it calls weight it only means weight if full buffers is zero. Otherwise, it just means decrement. Does that make sense? The words weight and signal are really, I think, were adopted with the binary lock metaphor in mind. I also hear when the thing is really a lock, I'll hear acquire and release as the verbs. Some versions of thread libraries actually define a lock type that things that lock acquire and lock release, which are really just the wrappers for semaphore weight and semaphore signal with the understanding that they're protecting ones and zeroes, okay? But it's not like this thing has to wait on that before this thing is allowed to signal it. Okay, sometimes you arrive at the bathroom and the door is open already. Okay, it just happens. Make sense to people?

As far as I have a single writer and I have a single reader, if you have – I won't write code for this but I'll just let you think about it. A lot of times when you are loading a web page, the HTML is often being sourced from multiple servers. Okay, maybe the primary HTML is being served from I don't know, facebook.com, but all the images actually reside on some other servers that are residing elsewhere, okay? And so all the information is pulled simultaneously.

Okay, imagine the scenario where you have not one of these things right here but you have three writers and they're all kind of competing to write and prepare the information that's actually sent over the wire to the single client. Well, then all of a sudden, you would have to – you would still want to use this empty buffers thing but then you would have to have an extra global, okay, or either that or something that is declared in main and shared with all the writers, so that they can all agree on what the next insertion point into the Internet is. Do you understand what I mean when I say that? And you have to have some kind of binary lock in place to make sure that no one, no two threads try to write to in this race condition matter to the same slot in the Internet or some piece of data will be lost, okay? Does that sit well with everybody? In the scenario where you have multiple readers consuming the information, maybe it's not really a web page but maybe it's just content that's being sent up from an FTP server and it's being handled by threads and that actually takes several different files simultaneously. You could use the same exact type of thing for the readers. Okay, does that make sense? Okay, so there's that. Let me work on one more example. This is a canonical problem that I think appears in all courses that teach concurrency. This next example is what's called the dining philosophy problem, where it's kind of a ludicrous setup, but nonetheless it is the setup. So, at the center of a table, I've drawn this picture 18 times now, there is at the center of the table you're looking down from the heavens at this table which has a plate of spaghetti at the center. Okay, and surrounding the table are five philosophers with their big brains, okay? And, in between each philosopher is a fork and every single philosopher follows the same formula every single day. They wake up, they think for a while and they eat and they think for a while and they eat and they think for a while and they eat, and of course they think for a while before they go to bed. Okay? So, there is four think sessions [inaudible] by three eat sessions, but the interesting part and philosophers are actually confused by this idea that they actually need to grab both forks on either side of them in order to eat the spaghetti. So, they are these geniuses who have to eat like this, okay? If I model each of these philosophers as C threads, so each one follows the same think, eat, think, eat, think, eat, think pattern, it might be the case that they both decide to stop thinking.

Two neighboring philosophers decide to stop thinking at the same time and will eat but I can tell you right now that this one and this one will never be eating at the same time, because they both demand the fork in between them to be in his hand, in their hands before they actually eat spaghetti, okay? But let me not worry about that, let me just write the code and pretend that everything just works out and that will illustrate the deadlock problem. Okay, I want to model the forks as an array of semaphores and I'm going to write them down this way. This is shorthand. You actually would have to prepare them with five calls to semaphore new. But that just means that every single fork initially before they all wake up, there are five forks sitting on the table waiting to be grabbed by philosophers. Okay, and that's all I need. I'm actually going to write a thread, one philosopher, and each philosopher knows where he sits around the table, okay? And so the formula I want them to all follow is the following: four, oh, I can't use I there, I'm going to use ID. My first stab at this is this right here where they think for a while, just assume that that's some thread safe function that doesn't require any kind of interaction with other think calls, and then they want to eat but that isn't going to happen. There is actually a think call down there but that's not that important. What has to happen is that the four of the philosopher calls in his thread space, he needs to acquire fork I and fork I plus one, being sensitive of the fact that fork I plus one may be fork of zero is I is really high. Okay, does that make sense?

So, what I want to do is I want to semaphore weight on two different forks. Forks of I and forks of I plus one mod five, if a philosopher is able to pass through those two things then he just realizes brilliantly that he has two forks in his hands, so he is permitted to eat and call this function. He is a polite philosopher, so that when he is done eating, he is aggressive about putting the forks down by calling semaphore signal of I and I plus one mod five, and it is perfectly possible that if I spawn off five of these functions in main. I set up the forks full array of semaphores and it is four loop from I is equal to zero up to but not including five, I call thread new and I pass in zero, one, two, three and four, so they all have an ID number between zero and five and everyone has a unique number. It is possible for this to work. Okay, some philosophers think more enthusiastically about a problem before they eat. Some just really – they have like I don't know philosophy block and they actually – and they want to eat right away but this is technically programmatically encoded into this as a possibility. I am philosopher number zero and I thought for a while, but I would like to eat now. And so I do grab fork number zero and I'm holding fork number zero and then I get swapped off the processor. Okay, make sense? This as a resource is not available.

That first zero, first one is now decremented to a zero at the time that the thread is swapped off. Make sense? Maybe the processor gets, I'm sorry, philosopher, the second philosopher gets the processor next, thinks for a while, gets this fork, okay, gets swapped off the processor. And then it happens again, and then it happens again, and then it happens again and then maybe this one wants to grab the right fork and it's like oh, can't do it because it's blocked. Okay, it's semaphore weighting on something that isn't available as a resource. So, it gets swapped off the processor, right? Maybe thread two, philosopher two says oh, I have the processor again, I'm going to grab the fork. Oh, I'm blocked. Okay, everybody is holding a right fork that is somebody else's left fork. So, when this slightly more obscure way, I think it's still pretty clear though, you have this mutual deadlock among all five threads because every single one of them is waiting on some resource held by the philosopher's to his left. Okay, does that make sense to people? So, you have deadlock. We can certainly overcome it. It's not a disastrous problem. There are multiple ways to solve it. You could actually alternate and have philosophers alternate between whether they grab the odd indexed fork first or the even indexed fork first, that could help solve things. But, I'm not going to take that approach. I'm going to take something else. Somebody had a question? Yeah?

Student:[Inaudible]

Instructor (Jerry Cain):Oh yeah, I'm sorry, it should be. This is just three iterations. Every single one of these should be IDs, that is correct. Sorry about that. So, I have to let you go in a minute, but I can really just communicate the problem I think pretty clearly. Remember earlier I said I wanted to do the minimum amount of work to prevent deadlock?

Okay, I want to implant the least amount of functionality to make something possible while still maximizing the amount of thread, amount of work that any particular thread can do during its time slice. Well, I could actually make this a critical region and that would be really rude because that would require that at most one philosopher can eat at any one moment, and that in itself is rude because you're not supposed to eat while others cannot.

Okay, but that isn't really the problem. What you could do is say you know what, I actually know that given that there are five forks and ten forks need to be held in order for everybody to eat, I can tell that at most two philosophers are going to be allowed to eat at any one time. Because if three philosophers are eating, that requires six forks to be held and we just don't have that many. Okay, we could also say that as long as we just prevent one philosopher from grabbing any forks whatsoever, it's technically possible to let four philosophers grab forks. Three of them may be blocked, but since I'm only allowing four philosophers to grab forks, exactly one of them will be able to grab two forks as opposed to one, okay? Does that make sense?

Now, two being allowed to eat, allowing four to try to eat, there are two different heuristics for solving the deadlock problem. I think both of them are very clear, are wise heuristics to go with. I will in the parting comments – I will tell you that I will at the handout take this approach. None allowed to eat, and that it's initialized to four. Okay, and the idea is that you have to be one of the graced four who is allowed to grab forks in the first place and try and semaphore weight on them allowed to eat right there. Most of them will be able to pass through it immediately. The only philosopher that would be blocked on a call to semaphore weight on none allowed to eat right there will be the one who is the only one who hasn't tried to start eating yet. Does that make sense?

After you release the forks, you could call semaphore signal right there on none allowed to eat. You may think it's really weird to allow four to try but all I'm trying to do is remove the deadlock. Okay, and I technically will remove the deadlock if I limit the number of philosophers trying to simultaneously eat, to not be five, but to be four. Okay, does that sit well with everybody? Okay, I will go over the actual code for that on Friday, okay, and then I will give you some more intense examples than that. Okay, have a good night. I will –

[End of Audio]

Duration: 51 minutes

ProgrammingParadigms-Lecture17

Instructor (Jerry Cain):I have one handout for you today. I expected to have two but we had photocopier drama this morning so I haven't photocopied your assignment yet.

I'll post it as a handout after class today. You have plenty of time to do it. I'm not gonna make this due until two Monday's from now. So it's not like you're in any immediate rush to get started on it, so that's why I wasn't stressing about it. But nonetheless the assignment will formally go out today and you'll have two weekends to do it.

It's actually not too bad. It certainly involves some of the new threading stuff, you're revisiting Assignment 4. You're implanting some thread directives to make a lot of stuff that was running sequentially before and you felt the pain of that sequential execution before. Make it so that lots of things run in parallel and things run really, really quickly, really, really beautifully. It's very impressive, it's a good end result once you actually get that thing up and coded.

When I left you on Wednesday, you were all stressed because you had a midterm in seven hours. But I was just getting through the dining philosopher's example. Let me review the picture. The idea is that there is food, a big plate of spaghetti, at the center of a table and there are – I'll be less caricature about it this time – there are four philosophers – and I'll put P0 there – sitting in a circle, and there are actually what look like pitchforks but just regular forks in between each of them.

And each of those philosophers is designed to run in his own thread and they follow the recipe, that is, they think, they eat, they think, they eat, they think, they eat, and then they think again, and then they exit as a thread. That is their day. In fact, that's their entire life because they only live for one day, okay.

So I set this up as a global resource, because that's the way the handout deals with it. I will emphasize again that the way I'm writing this – this is shorthand, all I'm really saying here is this is a gesture to the fact that I have five global semaphores and an array of link five that are all initialized to one. There really are some semaphore new calls that are involved to build that thing up but I don't want to focus on that other than just mentioning it.

Each one of those ones basically represents the availability of a resource. If I call this Fork 0 and this is Fork 1 and Fork 2 and Fork 3 and Fork 4, when this particular semaphore is locked up in this synchronized manner at a 1, I know that this particular fork is available. And that is of interest to Philosopher 1 and Philosopher 2 because both of those have to contend for that fork in order to pass into the eating phase of the thread, okay.

So without concerning ourselves with the deadlock scenarios that was outlined last time, this is basically the thread I want them to follow. Each philosopher knows their subscript or their ID number and that influences which forks they grab, or they try to grab.

And so if – where did if come from? For N to I is equal to 0, I less than 3, I++, I want each philosopher to grab the fork to the right, fork to the left, that grabbing happens via two semaphore wait calls against forks of ID. I messed that up last time, I wrote I, forks of ID and then forks of ID + 1 but we want to modify 5 just so it wraps around. And let's – there we go, there, okay. And as long as they pass through those and return from those two semaphore wait calls in sequence they're allowed to eat. This is after they think for a while.

Okay, and after that happens, they eat for a while, they semaphore signal these things. I don't really care about the order in which they free them. It doesn't really impact execution in an interesting way. Forks of ID plus 1, mod 5 and that is more or less what they do. There's an isolated think after the last meal and then they return.

And I did my little choreography last time where in principal because each of these five threads may be running in round robin fashion, they probably will be, each one might get to the point where they grab the right fork. They're enough through this call that they've effectively acquired the fork because the 1 has gone done to the 0, but they get swapped out sometime between the actual demotion there and the time they actually demote the 1 to a 0 inside. Does that make sense?

So it's possible, I don't want to say it's unlikely, it's actually unlikely unless you force it happen but actually I don't want to say that. Because it's possible it's a problem. If all five threads could be swapped out right here and all be experiencing mutual deadlock because they're all depending on the philosopher to his or her left to be releasing the fork that is their right but your left fork, okay. Does that make sense?

Normally what you do, put all your semicolons in, but normally what you do is you implant the minimum amount of work to remove the possibility of deadlock. We made some observations in the final ten minutes of lecture on Monday – or Wednesday rather, where we know because of 5 div 2 is 2 that at most four forks can be held while philosophers are eating, okay.

So we could recognize that there are at most two philosophers gonna be able to eat at any one time so we could actually put something out of the generalized counter right there and right there and actually force each philosopher to become one of the blessed two that's allowed to go ahead and grab forks. Does that make sense to people, okay.

So what I did is I did this semaphore, a single semaphore, none allowed to eat, and I could initialize it to two. Let me get rid of these arrows. And I could sneak in a call right here where I say, "Please wait for me to be one of the two that is allowed to eat."

Okay, it's something of a critical region. It's not the same type of critical region we saw before. Critical region normally means at most one because they are concurrency – there are race condition possibilities. This is a different type of critical region. In fact, most people wouldn't call it a critical region but I'll call it that. But we only want two threads to be somewhere between here and this part right here, okay, because if we have any more than that then we're nearing the scenario where we might have deadlock, okay. Does that make sense to people?

So let them go ahead grab forks, eat, release the two forks and when they're doing saying basically, "I have left the table." So that means I should signal all the other threads or all the other philosophers that might be waiting to – for permission to grab forks and do a final semaphore signal call right there, okay. Does that make sense?

Now, if you push two there I would completely understand why you did that. I personally, even though I recognize that there are at most two threads allowed to be in there, okay, or the way we've actually programmed it up that at most two philosophers will really be calling the eat function. My preference, just because I'm a purist about it, is that this be a 4 instead. Okay? Now a really terrible value would be 5. If I have 5 philosophers then basically the semaphore wait is just this gratuitous semaphore that everyone has to consume but there will always be one there.

The reason the 4 works is because as long as somebody's prevented from grabbing either fork then there's at least one philosopher thread that's capable of grabbing two forks. Maybe the other three or blocked, okay? But it's always the case that exactly – at least one philosopher will be able to grab two forks. Does that make sense to people?

That's the minimum amount of a fix that I need to implant into the code to make sure I don't have deadlock. So concurrency and multithreading purists usually like to do the minimum amount of work to prevent deadlock. There is a reason for that, because the minimum amount that you implant there – you remove the deadlock but you still grant the thread manager as much flexibility in how they schedule threads as possible, okay.

When I put a 2 there I'm taking more control over how threads are scheduled. That means up to three threads can block on this line right here as opposed to just one. Does that make sense? Okay, if you make this a 4 that means that up to four threads can make as much progress as is really possible before they're blocked and pulled off the processor, whereas if I make it a 2 we're blunting some threads prematurely, okay. Does that make sense?

Okay, so that's what I liked about that and this is why I prefer the 4. If you put 2, 3 or 4 there it will be programmatically correct. If you put 2 or 4 there I think I'd be correct from a design standpoint. I prefer the 4. Yeah?

Student: Are there – I mean, maybe this is a dumb question but why would we use a semaphore as opposed to, like, an if statement on the global variable? Is there any really good reason to?

Instructor (Jerry Cain):We could. In fact, I'm gonna talk about that specifically. You could – rather than doing this where you actually have the integer. Basically this tracks a resource. I like to think of this as the number of shopping carts that are available in front of Safeway, okay. And if there are five people that approach the store and it's a requirement that all of them have a shopping cart, they all flash mob the four shopping carts but only four walk away with one. So one is blocked until at least one shopping cart comes out the exit door. Does that make sense?

I could – and I think this is actually a really good question so pay attention to my beautiful answer here. That I could have just put a global integer here and said it equaled to 4. I could have had this check to see whether or not it was greater than 0 and if so, acted on it, but then I have the separation between test and action that was problematic in the ticket agents example. Does that make sense?

Well, you could solve that by having an exposed global integer and a binary lock like we did for the ticket agent's example but then what happens – and you have to think about this. I don't really want to write any code because I don't think I have to.

Rather than just blocking on this semaphore and letting the thread manager decide when you can get the processor back, what you'd have to do is you'd have to do some little while loop, okay, around and repeatedly check to see whether or not the global variable went positive from 0 if you were blocked on it. Does that make sense?

You'd have to keep acquiring a lock and releasing it, acquiring the lock and releasing it because you can't check that variable unless you actually are the only one looking at it. And – I'm losing my train of thought here. Where'd it go? The problem with that from a programmatic standpoint, and this is I think is a pretty good point, is that if you're basically while looping you really aren't allowed to do anything meaningful until you get beyond the while loop.

So what's gonna happen is you're gonna be hogging the processor, maybe in the same time slice you're gonna just keep reacquiring and releasing the lock just to confirm that it's still 0. Does that make sense?

That's what's called busy waiting. There are some scenarios where busy waiting is actually fine. It usually is in the case where you have multiple processors and you expect some other thread running on another processor to release a resource pretty quickly. But in a single processor environment, which is what we're pretending we have, there's no reason for a thread to spinlock and keep checking the value of a global variable because it's not gonna change until somebody else gets it. Does that make sense?

So this right here is this very clean way to say, "You know what? I'm not making progress. Let the thread manager pull me off; put me on the blocked queue. Only when someone signals the semaphore that I'm blocked on will the thread that's blocked on this thing ever be considered for the processor."

The alterative is to use the global – exposed global with binary lock, programmatically correct but it encourages busy waiting and busy waiting is like the – probably the – I don't want to say the worst thing, the worst thing is having, like, a race condition exposed, but it's as far as correct coding is concerned it's the least good in terms of design because it wastes processor time that could otherwise be shared with people who – for the threads that really can get the work done, okay. Does that make sense?

Two second question, four-minute answer. Okay, you guys are good with this right here, okay. There's that. I want to start talking about a couple other things. I have two related examples. One I'll actually write code for and then the other one I'll just talk about the design.

I want to start thinking about more practical problems. I'm gonna frame the next example in terms of just FTP downloads, okay. I know FTP is kind of this 1990's technology, but we all still use it, okay. We actually go and fetch files. We actually use programs that use FTP, but let me just assume the existence of this function.

I have this function and I'll call it download single file. And what I'll do is I'll give it two things. I'll give it a const Rstar. I call it a server and I'll give it a second name called path. I know you know what this means. Server is basically about the computer that's hosting the file that you're trying to get and path is basically relative to where the web directory or the FTP server is behind the scenes, how to get to the particular file of interest, okay.

So this is basically the computer it lives on and this is effectively the file name with directory structures, nested directory structures funneling down and drilling down to where the file lives on the server, okay. The return value here is the number of bytes that we're downloaded to pull the entire file over. Does that make sense?

Okay, what I want to do is I want to write this function called download all files. I want it to return the total number of bytes that were downloaded as basically the sum of all the sizes of the files that are downloaded. I'll call it download all files. I'm gonna assume all the files are hosted on the same server.

Okay, but I am going to give you – that's Rstar, the files an array of files on that server, and the number of files in that array. Now, two weeks ago – or a week ago for that matter, if I asked you to write this certainly on an exam you'd be dancing a jig because it's just a four loop, okay, with a plus equals going up and building up a return value.

But in spite of the fact that it's the same computer there's some problems with that. But just pretend that the server is capable of hosting as many simultaneously requests as possible, okay. Does that make sense? What you'd rather do is you're rather spawn off N different threads, okay. Does that make sense? N threads where each one is dedicated to pulling over one of these files, okay. Does that sit well with everybody?

Okay, so I'm gonna assume that run all threads has already been called and this is running as part of some thread that was spawned in the main function, okay. So I'm already dealing with a multithreaded environment. What I could do is I could declare something called total bytes and said it equaled to 0 and I can be prepared to return total bytes.

You're not necessarily sure how that's updated yet but you can be sure that each of the N threads that I'm gonna spawn off to download a single file is gonna somehow do a plus equals against this thing right here, okay. This is functioning somewhat as a num tickets in the very first example except we're adding to it instead of minus minusing from it, okay.

Now thread new doesn't return anything, okay. So what has to happen is that I have to spawn off N threads in addition to passing the server and one of the file names to the thread, okay, so that the thread can call this function.

I also want to pass the address of this integer right here, okay. Does that make sense? And a lock that's designed to protect access to this because you're gonna have several threads trying to plus equals potentially at the same time, okay.

So I'm gonna do this semaphore, I'll call it lock. I'm gonna set it equal to 1 and that's – this is just shorthand for what would really have to be there. And then I'm gonna do this, 4 int I is equal to 0, I less than N, I ++. What I'm gonna do is I'm gonna call thread new. I don't care about the debug name but I'm gonna call this function called download helper.

You may ask why I don't call this function directly. The function that's passed right here to thread new has had the void return type. But even if it didn't have a void return type there'd be no way to actually catch the return value from the thread new environment. Okay?

So what really has to happen is I have to call this proxy function that sits in between this one and that one that knows to call this one and also to accept its return value and do a plus equals against this thing right here, okay. I have to pass in a few arguments. I have to pass in server. I have to pass in files of I. I should pass in address of total bytes and I should pass in lock. That means I should pass in four parameters, okay. And that's all I want.

Now there is a problem with this, the setup already, but I'm just gonna implement it like this is the okay thing to do. But conceptually you know what's happening here. Basically this thread is being the typical manager at a company where he doesn't want to do anything except delegate, okay.

And this thread has the responsibility of pulling into these files. It happens to have N employees or N people it can hire on an instant like it just does right here. And it gives each one of them a task of downloading each of these things in succession. Does that make sense?

This download helper has to be a function that returns void. I'll call it DH for download helper. And it takes these arguments, const car star server const car star path int star, let's say, num bytes p, for pointer, and then it has this semaphore I'll call lock.

Now the semaphores, remember, are pointers themselves so they don't have to – you don't have to pass the ampersands there, you can just pass lock itself, okay. So you have a hook on the master copy of the integer that needs to be plus equaled against. What has to happen is that you want to do this, let's say, bytes downloaded and you wanted to clear that ahead of time because you want to do – actually, I don't have to do that. I'm sorry.

And you want to set it equal to the result of that function. Download single file where you pass in server and you pass in pass. It looks like it's being done sequentially but it's not. There are several there – there are N minus 1 of the threads trying to call this same exact function to download the files in parallel, okay. Does that make sense?

It's in the same spirit as the type of thing you have to do for Assignment 6, okay. The reason you catch the return value is because after you let this thing run and do its work, as a side effect of this function you're just supposed to assume it's on file and full appeared on your host machine.

But afterwards what you want to do is you want to semaphore wait on the lock and all the other threads are quite possibly waiting on because once you acquire that lock you are the blessed thread that's allowed to plus equals against the single integer that you have a pointer to.

So num bytes p plus equals bytes download. Then you go ahead and you release the lock and then you return. There's no explicit return value here. The thing that feels like a return value is actually a return value via reference via this point that you have a side effect of downloading the one file you're supposed to and you're also supposed to update this variable to be that much higher so that when the thing is returned it returns presumably because all threads have returned and contributed to this thing, okay.

So when this happens it really is the accumulation of all the bytes that have been downloaded, okay. Does that make sense? Okay. If I'm silly and I accidentally acquire the lock before I call download single file it'll still return but it'll be even slower than it would have been had you just not used threading at all and just done the download single files sequentially.

Because this just means if I put this up there and this is serving as a binary lock then you're putting the very important time consuming function inside the critical region so that at most one thread can be involved in the downloading of a file. Does that make sense?

So it's imperative that this be outside the critical region and we only have one critical region that it has to come after because it has to appear somewhere it has to come after you download the file, okay. Does that make sense? Okay, now there's one problem with this right here. Do you have a question?

Student: If you had, like, 10,000 files to download would it make all 10,000 threads at once?

Instructor (Jerry Cain):It will in principal it would. That's a scalability issue that I'm not concerning myself with. I'm assuming that we're dealing with something like 40, okay? Or even 100. Most thread systems, including our own, actually has a limit on the number of threads that can be reused. I'm sorry, that can be spawned off.

Our system is like 500. Some systems it's like 1000. And in principal really sophisticated systems can actually reclaim threads that have completed and reuse that thread space for something that's created afresh, okay. Does that make sense?

There is a little bit of a problem with this. I've framed this in a way that might not make sense to you but I just want to make sure you understand it is that I'm assuming that run all threads has already been called and this is a function, it's actually running in some of the thread.

And so what I really want to happen is this is a child thread of main to really be spawning grandchildren threads, okay? If run all threads is already – has already been called and this is running, as soon as this is called it actually sets this up for all – or all N of these up on the ready queue immediately so that they start running immediately, okay.

I used the analogy Monday, I think, of a dog that's already in the race giving to N dogs and throwing them back to the beginning of the race and letting them run, okay. The problem here is that the way this is coded up the job here is actually very easy. This isn't the equivalent of the manager of a company who's delegating all of his work basically going home before the work gets done. You understand what I mean?

It's one thing for you to delegate work and then to go home for lunch and then never come back. It's another thing for you to delegate the work but then to wait for all of it to be done even if you're just in your office surfing the Internet. You can actually just – you should hang out until all the work is done so that you can properly return the total bytes value, okay.

The way this is technically coded up at the moment it says, "Okay, I'm gonna declare a local variable. I'm gonna – yeah, yeah, yeah, I'm gonna share a lock with all of my underlings over here so that they all have atomic synchronized access to the shared global right here."

And I spawn them off and then I return immediately. It's quite possible that I would return a 0 here because I may advance from the very last iteration of the for loop to this return statement. Okay? Does that make sense? Before any one of these threads makes any partial progress whatsoever.

Presumably this is a time consuming function, like, on the order of, like, milliseconds or even seconds. It would take milliseconds for it to advance from the very last thread to this one right here. Okay? It's quite possible that maybe even one or two time slices all of these threads could be spawned off and it could return before that makes any work whatsoever.

So what really has to happen is right there I really need the manager thread, the download all files thread, to really block and not go anywhere and certainly not advance to the return statement until it has verification that all of these guys have completed. Because if they've completed then the manager knows that this thing has really been built up to have the true return value. Does that make sense to people? Yes? No?

Okay, so what do you do? Well, not surprisingly you use concurrency and you use semaphores. So basically implant thread communication, thread-to-thread communication. The reader-writer example we had the reader and the writer communicating in this binary rendezvous way. When I do this I'm talking about the crisscross of semaphore signal and semaphore wait calls so that each one can tell the other one that there was an empty buffer or a full buffer, okay.

There was this one to one relationship between reader and writer there. I really have a 1 to N relationship with this setup. I have a single master thread right here that's supposed to do all the work. It elected to spawn off N threads to get the jobs done because it can take advantage of parallel computing right here, okay, and download many of the files simultaneously.

What really has to happen is I need about six inches more of board space. What I want to do is I want to declare a semaphore up here and I'll say a children done and I'm gonna set it equal to 0. I'll do the same thing there just to make – there really is a semaphore new call.

What I want to do is I want to use this children done semaphore basically as this connection to each of the threads that it's spawning off. I wanted to do this for hence I is equal to 0, I less than N, I ++. I wanted to semaphore wait on children done.

Now I haven't passed children done to that thing yet, but I will in a second. What I want to do instead is to change this to a 5. I want to pass down children done as an extra parameter. I have to abbreviate because I'm out of room there, okay?

So what I'm basically doing is I'm giving, like, it's almost like a little baby monitor to each of the threads, okay, that I'm spawning off. And when each one of them is done they go, "Wah," into the baby monitor in the form of a semaphore signal and when I hear N of those, okay, I know that all of the threads have completed. Does that make sense to people?

So the signature for this has to move over a little bit, semaphore, I'll give it a different name here. I'll call it parent signal and then before I return over here I will semaphore signal the parent to signal. This is the "wah" into the baby monitor and this thing is actually aggressively for looping, okay, and a semaphore waits on this thing not once, but N times, once for each of the threads it's spawned off.

Programmatically each thread signals this thing exactly one time. So I expect this thing to be signaled exactly N times. I need all N of those signals in order for this thread to know that it's done. Does that sit well with everybody?

And then once I have that I can advance to the return statement knowing that it's safe to return total bytes there because all of the N threads I gave birth to have actually done their work and died. But as a side effect their legacy was to plus equals my total bytes integer, okay.

Yeah.

Student: Is there any way that there's a way – it seems like there's a way to increase the number of N semaphore and is there a way to decrease it [inaudible]?

Instructor (Jerry Cain):Well, semaphore wait certainly does decrease it. If this is surrounding a 7, semaphore wait brings it to a 6, okay. So semaphore wait is like a minus minus that's guaranteed to be atomic and it's also a block if the number inside happens to be 0. Semaphore signal is an atomic plus plus and not much more than that. Okay?

Student: So you could initiate it to N semaphore children have not done, I guess? And then [inaudible] until the lock is –

Instructor (Jerry Cain):Right, but I have the – our version of the semaphore, some more recent libraries have decided to do this even though this is an argument against it, you'll notice – and this is mentioned in the handout. We don't provide a get value within method on the semaphore. We could ask for the value of a semaphore, okay. And we could say, "Oh, what is the value now?" And it tells you 7. And you go, "Oh, I'm gonna act on that 7." Right?

But it may have changed by the time you get a chance to act on it because in theory if you have a lot of threads the time between the return of get value within and the code that acts on that value could be separated by seconds, okay. Exaggerate it. Think about it in terms of years, which is what it really effectively feels like at the processor level, okay.

A lot can happen in years, okay. So you don't necessary – you can't trust a value that was acquired from within the semaphore and act on it until you've actually released a lock on the check, okay. Does that make sense? Now actually getting a 0, there are some situations where it would be okay. You could keep on looping and only break out once you get the value out that's a 0, but that's a busy waiting thing that I was arguing against when I answered her question earlier. Does that make sense?

Technically this is a little bit of busy waiting but it makes as progress as is possible until it blocks because not enough children threads have completed. There are some versions of semaphores. The one I'm thinking about are the ones that come with the 1.5 version and later of Java, which you'll learn all about in the autumn when you take 108.

Do you notice the semaphore wait right here, it's an implicit request to just do a minus minus; a minus equals 1, right? There are some flavors, not in our library but some more modern libraries where you can actually for a total of, like, N or 12 or 20 or 3 dozen or whatever, decrements against the semaphore and just call this once as opposed to exposing the for loop as an internal for loop instead. Does that make sense?

Okay, ours it's exposed. I think that's fine at this stage of the game because I want you to understand the mechanics of what's involved in order for this thread to block, okay. And it's not technically busy waiting so it's not really a bad design, okay. Make sense? Okay, question.

Student: Yeah, there is nothing to prevent all the threads from downloading into the same portion of –

Instructor (Jerry Cain):I didn't – not the same portion of the file. My assumption is that each of these files is actually different. And I'm just assuming – I didn't mention it explicitly but I'm assuming the download single file is the thread save function that doesn't actually interact with other calls of the same function at the same time, okay, which is technically not true because two download single files may have to create at the same directory on the host machine, and that would be a little bit problematic except that the make directory command is implicitly atomic on the operating system anyway.

I'm sure it is. I've never read that but I'm sure it is, okay. You got the setup here? Yep.

Student: Is there any reason not to, like, in [inaudible] when you secrement it there and just semaphore wait once at the end?

Instructor (Jerry Cain):But the thing is – if you semaphore wait you might wait on – you want it to semaphore wait after it's become 0. There's no way to do that. Like, semaphore wait will succeed if it's positive and you're making it initially positive. All you're gonna do is make it N + 1, okay, or take it from 0 to 1 at the end. But it's, like, you actually want it to – the only way you can block if it's semaphore waiting on a value of 0 and you really do want this thing to halt, okay. Does that make sense? Okay.

Some version of the library, and actually we can't initialize – even though a semaphore in our world can never go from 0 to a negative number, some versions – the Java library does this as well, allows you to initialize a semaphore with a negative number, okay. That seems a little weird but if I were to initialize this semaphore to be negative N + 1, and then I had a single semaphore wait call here, it would have to be signaled N times before it became positive. Does that make sense?

So that's what I think the two of you are trying to get at. It's just not supported by our version of the library or our particular thread library. Yep?

Student: So you can [inaudible] the function together you make it one of the semaphores like with positive have a [inaudible] of counters and some of the counters at the end?

Instructor (Jerry Cain):I'm sorry, what? Oh, I see what you're saying. So in other words you're thinking, like, some how do a –

Student:[Inaudible] or is it there's an each [inaudible] that's in reference to one [inaudible] so you don't really need to have a global counter.

Instructor (Jerry Cain):I'm still – I'm not quite understanding. I think I know what you're saying. Are you just asking this thing to pull, like, an array of Booleans or something like that?

Student: That's [inaudible].

Instructor (Jerry Cain):Into an array, yeah.

Student: Of elements?

Instructor (Jerry Cain):Right.

Student: Each?

Instructor (Jerry Cain):I see what you're saying, yes. I have seen that before. What he is suggesting, I think it's pretty clever although I don't – I think it's clever but I think it's just a little bit more work than it needs to be. He's setting aside not just one integer but an array of length N where each of these things can write without fear of race conditions to a slot in the array that's dedicated to that thread.

And then once you get past this point you'd still need the children's done semaphore, you can go through and safely do all the addition yourself there. Right? That's actually a fine idea. I actually – I can't even say that it's a bad thing. In many ways it's actually pretty good. The only thing about it that it might be problematic is that, I mean, the number N here is huge. But even that's not that big of a deal. So that's a great idea. Yep?

Student: Reference you back to the question two questions ago, if you made children done equals N and you replace the for loop with while children done does not equal 0, would it work then and just like a closed while loop?

Instructor (Jerry Cain):Well, you can't do equals 0 on child's done. I'm writing this as shorthand but it's not really an exposed integer. This one doesn't actually – if the semaphore did provide a get value of method or function there are a lot of scenarios where that method – that type of functionality breaks down. But this would be one where it would work because it's only gonna hit 0 once.

So that is true but it's interesting to spoil that. If you want, like, after you've done this Assignment 6, you should go – you already know enough Java to digest the syntax, just go and read, like, the two or three pages of the concurrency model in Java 1.5. You're gonna read it anyway if you take 108 in the autumn, just so you understand all the things that are available and more sophisticated for our libraries, okay.

I just wanted to minimilize I wanted to. Basically we've used this package for so long because it really is lean and clean and very easy to digest in full. So you have to do all the interesting things in terms of the atomics, okay. You guys doing okay?

Okay, I do want to make it clear that when I initialized this to 0 I may get down here and I may spawn all these things off and in theory semaphore wait against a 0 and block immediately. Does that make sense? I may actually get swapped off right after this closed curly brace but not get inside the for loop immediately. I could get swapped with the processor as this download all files thread.

It might be the case that 50 percent of these things actually complete and promote this from say 0 to 6 because there were 12 child threads, okay. And this thing would come down right here and it would happen to notice – or not notice but it would benefit from the fact that this thing had been promoted 6 times. So it would make 6 immediate for loop iterations – or carry through all 6 of the iterations immediately before it's blocked on the seventh.

It's not a problem, okay. It's technically possible for the download all files thread to exit before the download helper thread does. I'm sorry, before any of the 12 download helper threads exit. Now you would think that'd be a problem but this is the scenario. This one blocks the 12 different threads that I spawned off all get this far and they get swapped off just after they've returned from semaphore signal here but before they actually call the return instruction to emulate the closing curly brace. Do you understand what I mean when I say that?

So all swapped threads could, like, right before here be swapped off the processor. And this thing could get the time slice after that's happened; get the processor back and say, "Oh, wow. Look, I actually went through this thing 12 times and I can return an I do." Okay?

You're not – I mean, if you want to be as completely realistic about the emulation as possible you can do it. But you have to effectively understand that this function is really over when SS – semaphore signal returns. And that to the extent that you needed the information to flow and be synchronized the way it is it still works, okay. Does that make sense?

In Java you actually have the option of releasing the lock after the closed curly brace. There's a key word in that language that doesn't exist in C and C++ called synchronized. You'll become very familiar with it next autumn, that releases the lock after you've formally finished the method. And I say formally finished after you've gone past the closed curly brace, okay.

But I don't care in this situation because all I really use the semaphore for was to get a meaningful full total bytes value. Make sense? Okay, so that's – oh, go ahead.

Student:[Inaudible] blocks before the [inaudible], right?

Instructor (Jerry Cain):This one right here? Yeah. It's a one line body of a for loop, so just this is right here and like that. Is that what you're asking?

Student: Yeah.

Instructor (Jerry Cain):Okay, very good. Okay, so what I want to do is I want to set up a big example for Monday, okay. All of the examples have been focused primarily on either one or maybe two synchronization directives. The handout that I have you out today – gave out today – has this great first 2 or 3 pages that was written ten years ago but it hasn't changed. It hasn't changed in 50 years, really, much less the last ten years.

It describes all of the different patterns that you have to be sensitive to or situations you have to be sensitive to and the patterns you use to be sensitive to actually make sure that you don't have race conditions and you avoid deadlock. It's all about protecting global variables in a binary lock – or a critical regions situation and implanting directives for two threads to communicate with one other – two or more threads to communicate with one other, okay.

And so it's good stuff to read. I've hit on all of it in the past three days – or past three lectures rather. The large example and it's kind of – it's a little intimidating. It's actually larger than any exam problem. It's probably the merging of two past final exam questions from, like, ten years ago. It's a great problem, okay?

It is the ice cream store simulation and all I'll do is I'll set it up so that you know what all the players will be in this big grandiose ice cream store, okay. We're going to have ten customers, okay. There's going to be a customer function. We're gonna spawn off ten of those things and we're gonna actually know how many ice cream cones each of the customers order. It's gonna be between 1 and 4, okay?

There's gonna be a cashier that each of the ten customers approaches as they're ready to pay. So there's a little bit of contention and I think there's already clearly a flavor of currency and race condition going on because these ten customers have to approach this cashier and the best metaphor I can think of is when your mom, when you were going with her a little kid to the market she'd have to go to the deli and she'd have to potentially edge out and get atomic access to that little number thing.

She pulled 33 or the 94 or whatever it was, and you always looked at the number and it was like 22 away and you go, "Oh, my God, we're gonna be here forever." Okay? But there certainly is gonna be thread-to-thread communication between all the customers, okay, and this one cashier. Make sense?

There's gonna be a single manager thread – I'm being very cynical about managers but this manager doesn't do very much either. What he does is he approves or disapproves of ice cream cones, okay. He doesn't make them himself, he just approves and disapproves of them with a – like a coin flip is what really happens.

They're gonna be between 10 and 40 clerks. The reason it's 10 and 40 is because every single customer is going to give birth to a clerk thread – one clerk for every single ice cream cone that needs to be made. So the people who are ordering, they're really impatient, this is real life. They want – they have four ice cream cones to be made, they want all four to be made in parallel, okay.

So the customer never sees the manager but these ten customers interact with 10 to 40 clerks, order the ice cream cones, they accept the money, make the ice cream cones, but all the clerks interact with the manager. They have to acquire a lock on the manager's time, okay, because the manager is overwhelmed if more than two people are in his office at any one moment.

So he can only accept one person in the office with one ice cream cone so there's no confusion as to which ice cream cone he's approving or disapproving of, okay. Does that make sense? So we're gonna run a simulation where there are up to 52 threads running simultaneously and there's certainly gonna be 4 different types of threads, okay.

I want the manager to basically arrive at the store and not to leave the store until he's approved the number of ice cream cones that he needs to approve. I want each thread – the clerk, to only – to live as long as is necessary to make a good ice cream cone and hand it over to the customer. I want the customers to be able to order their ice cream cone, get it, go and pay for it, and leave the store. I want the cashier to know that he's rung up ten different customers. Does that make sense?

Okay, so to the extent that you can read the problem statement between now and Monday, that's great, although don't worry if you can't because I'll certainly review this at the beginning of Monday.

But even though parts of the simulation are certainly a little contrived, they're contrived specifically to bring out some scenario where you have to deal with concurrency properly, okay, and use semaphores to either foster thread communication or to manage binary lock or access to a resource, okay. You have a good weekend. I will see you next week.

[End of Audio]

Duration: 49 minutes

ProgrammingParadigms-Lecture18

Guest Instructor:All right, everyone, Jerry's out of town today so I'll be giving the lecture. And we have two handouts today, two very small handouts actually, the section handout and the solution for tomorrow.

So I believe last Friday Jerry started the ice cream store simulation problem. This is a really huge problem. It's probably bigger than any single synchronization threading problem that you saw so far in the lectures, and it's the merger of two or three final exam problems, so it's pretty involved. So try to go through it as clear as possible but feel to ask me any questions if you want.

All right. So let me just start from where we left off last week. So we were talking about the ice cream store and we have four places basically in the ice cream store. We have the cashier which is they are basically to handle the billing issues and we have the customers. We will say we have ten customers and each customer is gonna ask for a random number of cones – of ice cream cones from one to four.

So the least we could have it ten cones and the maximum is 40 cones. And every single customer because he doesn't want to wait for every single cone to be done sequentially, then he's gonna fight of a clerk or dispatch a clerk for every single ice cream cone that he wants. And so we will have clerks that are making the ice cream cones. We have 10 to 40 of those. But the thing is every single clerk because really giving the ice cream to the customer it has to ask for the manager's inspection to get an approval for the cone that he made.

So the clerks have to get the manager's approval, and the manger that is only a single manager, only a single cashier, and if the manager disapproves of the cone then the clerk basically has to redo the cone and ask again for the manager inspection. The manager, he doesn't like to have two clerks in his office at one time. So basically we can't have more than one clerk contacting the manager at some point.

At the same time – so this actually starts – you can start to see the communication that is going between those guys. We definitely can see that we have clerk threads, right? Because every customer – every single customer would potentially ask for one, two, or three, four clerks. At the same time, we have ten customers that are running in parallel. It's not only a single customer at a time. So we have threads here, we have a thread for the cashier, a thread for the manager, a total of probably 52 possible threads, okay.

You can see the communication going between the cashier and the customer. You don't expect any communication between the cashier and the clerks or the manager, right? You can see communication between the customers and the clerks depending on how many clerks he fires off, and you can see definitely a communication between the clerk and the manager, right.

So you want to start thinking about what are the constructs that you will need in order to maintain synchronizations issues between all of those players in the game, okay. So let's start thinking, just try to brainstorm about it and then we can look at the code. The thing is the customer comes in and he wants one to four ice cream cones and he will fire clerks. After he gets all his cones he will go to the cashier, right? But he cannot go to the cashier until all the clerks give him back his ice cream cones.

So you can start to see, like, this kind of 1 to N communication that we talked about, right. You can see that this guy is gonna request N clerks and he has to wait for the N clerks to get the job done before he can go to the cashier, right.

You can also see the communication between the clerk and the manager. So no two clerks can go to the manager at a time which means that there should be this kind of lock around the manager's office, right. And so only one clerk can access the manager's office. At the same time the manager will be waiting all the time because he has to wait for all the ice cream clerks to be done for all the customers before he can go home.

So the manager will be in kind of a loop just waiting to be requested for an inspection. And once he finishes the inspections he gets back the result to the clerk and he keeps waiting again until all the potentially 40 cones are done and inspected. Make sense?

So the clerk will basically have to request from the manager who's waiting. So there's the manager waiting for the clerk to request. There is also the clerk waiting for the manager to finish his inspection, right.

So we have this kind of binary rendezvous again thing going on in here. You see that? Now there is another constraint on the customers going to the cashier at the end. So the customers are waiting for their cones. And after they get their cones they just go to the cashier in a line and this line has to be maintained on a FIFO order, so a first in, first out, right. And we have to also think about that. How are we gonna insure that the customer that arrives first at the cashier gets really served first and leaves first, right?

You will see, again, the kind of binary rendezvous thing between each customer and the cashier because the cashier has to be waiting all the time for a customer to request the service or basically a customer to show up in line. And after a customer shows up in line he will be waiting for a cashier to finish handling his billing and let him go basically, right. So any question on how the setup is for the problem?

All right. So let's talk about the main function. Let's see what happens in the main. Okay, let me write them in here. All right. So we have the main function. I'm not gonna write this argument but it basically has to maintain the total number of cones, right? The main function is basically responsible for spawning all the threads that we need.

We'll see that the customers are responsible for spawning the clerk's threads but other than that everything gets spawned in the main function. So the main function is gonna basically – you have the total number of cones which is required to spawn the manager because the manager needs to know how many cones there is because he can't leave until he's done with all the cones, right.

So we do the normal stuff, we entered the thread package and we sent up some semaphore that I want to introduce right away. But just bear with me until I get to this point. And then we basically spawned all the customers. So for int I is equal to 0, I is less than 10, int I plus plus. What we want to do is we want to get a random number of cones for every single customer and then spawn the customer and add this number of cones to the total cones that are in here.

So I need to initialize this [inaudible]. So we have an int num cones is equal to a function random integer. That gives me back a number of cones between one and four and this function is in the more concurrency handout slide. I'm not gonna get it.

Now once we have the number of cones for a customer then we can just start spawning the customer because all the customer needs to know is how many cones he's gonna order, right. Make sense? So we got a thread new, debug name, the customer function, and this customer's gonna take one argument which is the number of cones, right. After doing that we have to add this number of cones to the total of cones because we ill need to pass this total number to the manager eventually, right.

So total cones plus equal num cone. Now this is gonna be done for all ten customers and this way we spawn all the ten customers that we have in our simulation. Now we need to model the cashier so we have a thread new, the cashier, and we don't give him any parameters. And we also spawn the manager so we need thread new, the manager, and we pass to the manager the total number of cones, which is single parameter, right.

Now we're done with that, we just run all the threads. After we run all the threads since we did set up the semaphores in here and semaphores, again, remember this is a construct that is appointed to something that is allowed in the heap. So you have to always remember to free your semaphores, okay.

So you go and free – that's called a function free semaphores and then you go and end all of the semaphores that we're gonna introduce, okay. And then you return zero. So this is basically your main function and I don't think, like, in here we're just spawning threads. You don't see any kind of communication going on. We didn't do anything yet, right? We're just starting.

So I was kind of thinking about where to start. Typically the sequence, the logical sequence is the customer because that's the customer that goes to the store basically to buy ice cream. But I think it might be really confusing to start with the customer. So I will go with the manager because it's the easier one and then maybe we see the communication between the manager and the clerk, right.

And then we try to link it to the customer and the cashier because the customer will also have to interact with the clerk. So let's go to the manager. The manager and the clerk. I just described a lot of things going on, right. A lot of basically synchronization things that we have to take care of. We need semaphores in here, right. There is like a [inaudible] I've just explained. We need to know after the manager inspects a cone whether it succeeded or not, right.

So that should be something communicated. It is some sort of communication. So because there is a lot of variables, there are so many variables and there are many threads or accessing all of them, we again do – we access them globally. So we'll have a global variables, global semaphores. There'll be a lot of them so in order to make it understandable we're just gonna group them in structures.

So we'll group all the variables that have to do something with a manager and a clerk in one structure and call it inspection and we'll later on do the same thing for the customers and the cashier. We have another structure with all the variables for the synchronization and we'll call it the line, all right.

So let's have our first global – so we have the struct inspection and, let's see, the manager's gonna inspect everything that the clerk shows to him, right? So we have to need basically a Boolean to show us whether the cone passed or not, right. And I'm gonna be very informal in here. I'm gonna give you the initializations but all the initializations are basically done in setup semaphores, okay.

So we're not gonna go through this function. But this pass is gonna be false. Now what else do we need? The manager has to keep waiting until it's requested, right, for inspection? And also the clerk has to wait until it knows that the manager finished its inspection, right?

So we need some kind of semaphore for rendezvous the semaphore is gonna be – let's call it requested. And so this is gonna be signaled by the clerk to the manager that it requests an inspection, right. Make sense?

All right. So this semaphore is gonna be initialized to zero, okay. So basically the manager what it's gonna do is basically gonna wait on the semaphore which means it's gonna block, okay. Until a clerk comes and sends the semaphore so it basically wakes up the manager. Make sense?

All right. So we have the semaphore requested. We have the semaphore finished and we also understand that once the manager – once the clerk requests inspection it will wait on this finished semaphore which means it's gonna block until the manager finishes the inspection and then signal the semaphore and then it wakes up, okay. So what else do we need in here? What else do you think we're missing? Okay, let's start working and see what we miss, okay. Let's start working on the function and see what we need.

So we'll have the function manager. So that's void manager, the manager takes an N which is the total number of cones. So let's call it total cones needed. And here is what the manager does. I'll have two variables, one of them is required, the other one is not.

The first one is total, let's call it num approved. So this is num approved which is initialized to zero and another one which is num inspected also initialized to zero.

So what we need to do is the following. While the number approved is less than the total number needed then you basically can go home, you have to keep waiting and keep inspecting, right? So while this is the case what you want to do is the following. You want to wait until a clerk requests your service, right? Okay, so you want to wait on inspection, let me finish this here. So you want to wait on inspection, not requested, right? Once you get a request – yes, go ahead.

Student: Which are not needed as [inaudible]? [Inaudible] not needed?

Guest Instructor:Those are not needed?

Student: Yes.

Guest Instructor:So did you see when that spawned the manager thread? We passed the total cones and the total cones were the total number that a customer's wanted. So that's the total number that basically the manager needs to inspect. That's total number of cones, yeah, that all the customers will buy, you see.

Student: Oh, okay. So that's the total number of cones.

Guest Instructor:Yeah, I'm just calling – I mean, I'm just changing the name of the parameter but that's fine, okay. Any other questions? All right. So you want to basically wait on the request for the inspection, okay. And then once you get a request for an inspection you basically want to inspect the cone, right? Okay.

So what you do is you have your – once this is done you basically are inspecting a cone so num inspected will go up and you have to check if you're gonna approve this cone or not. So we simulate. This is modeling, right? So we have a random function that says sometimes, yeah, we do approve, sometimes not, okay.

So we will say inspection not passed which is the Boolean in here is equal to some random chance. It returns one of the two things, zero or one. And again, I'm not writing this function but you understand what it's doing, okay. So again remember I'm assigning this to dot passed because this is the Boolean that I'm using to communicate between the manager and the clerk, okay.

Now if you really passed – if inspect got passed then what you want to do is basically increase the number of approved. Make sense? So num approved plus plus. Now we finished your inspection, okay. The only thing he has to do but what you know is that the clerk is potentially waiting on your finished. So you want to signal finished. Any questions? Yeah, go ahead.

Student: Yeah. Are we using global variables mean or –

Guest Instructor:Yes, we are.

Student: Okay.

Guest Instructor:So this structure is basically a new one.

Student: Okay.

Guest Instructor:Okay? And everything inside is basically initializing this function [inaudible] semaphore, okay. And I'm giving you the initialization. So all these are gonna be semaphore new, okay. All right. So after we do that what's left off is just a signal, the inspection not finished because you know that the clerk is gonna be waiting on the semaphore, okay. And that's your manager and that's the easies function of all.

Now let's go to the clerk. Now any question on that before I move on? All right, so the clerk. The clerk – we'll pick some parameter. I'll worry about it later. Because this parameter's gonna be rated to the customer and I didn't write the customer function yet.

So, let's see, what the clerk needs to do. It needs to make one cone and have the manager inspect it, right. It needs to repeat this cone until it passes, okay. And once it passes it just hands the cone to the customer. All right.

So, let's see, the clerk basically starts with a cone that doesn't pass. So Boolean passed is equal to false. And why you didn't pass you basically need to make a cone, right. And then request the manager to inspect it, okay. So let's see, we can potentially have up to 40 clerks requesting the manager to inspect their cones, right? Okay? Does this give you any hint about what we're missing here?

Student:A binary lock.

Guest Instructor:Exactly. So we need the binary lock. We basically need to lock the manager's office, right, so that we can insure at any point in time only one clerk can get in this office, okay. So here we need another semaphore which I will call lock, and this semaphore is gonna be initialized to one, right. Why is it initialized to one? Why not zero?

Student: So it can get in the office.

Guest Instructor:Exactly, so that at least one clerk can get in. If you initialize it to zero then nobody's ever gonna get in the office, right. So you want one clerk at least to get in. And you want only one clerk to get in. You cannot have it zero. Zero is a deadlock, right, because nobody's gonna ever be able to get in. You cannot have it two because two means two clerks can potentially be in at the same time, okay.

All right. So the first thing that you want to do basically is to acquire this lock, right? So you want a semaphore wait on inspection dot lock, and again, there are two scenarios.

First of all is that lock is still one because nobody did the wait before in which case you'll be able to proceed. The other scenario is that this is zero in which case you just block on it and wait until it is signaled, okay.

All right, so you wait on this lock. After you wait on this lock you basically need to request the service of your manager, right. Okay? So what you want to do is semaphore signal, inspection requested. Am I going too fast? Is it clear? Okay, so we signaled the request and after the – yeah?

Student: Some [inaudible] to one?

Guest Instructor:Excuse me?

Student: Would it say requested to one?

Guest Instructor:It would, yeah. So request is originally zero, right? The manager, which I just wrote in here, it doesn't wait on requested which means it blocks. So the way our semaphores in the library that you are provided the semaphore doesn't go to a negative value, okay.

So it always maintains a zero or a positive value. So if it's zero and you try to decrement it, it doesn't not allow you and it blocks. So you keep waiting there blocked until somebody makes it positive. So you can decrement it to zero, okay. And this is how the blocking works, right.

So what happens here is that the manager would be blocked if he starts before the clerk because what will happen is the manager will try to decrement the zero semaphore but it will block and then it will wait until some clerk comes and does signal the inspection request which will make it positive so the other one will be able to proceed and the command does. Do you see how this works? Okay?

All right. So you did signal request. Now after signal request the manager will get the request, right, and will start working on this. What you need to do at this point is wait until he finishes. Make sense? So you just –

Student: It's really [inaudible] is like if you have inspection request of him and if you lock up a manager why is it he requested?

Guest Instructor:That's a good question. Let me finish it and then I'll answer this question, okay. I'll just go on because, like, I want you to see the whole picture after we finish it. So we will basically wait until the inspection is done. So after the inspection is finished and once inspection is finished you basically want to know the result of the inspection, right? You want to know if your cone passed or failed. Now how do you know that? Which field tells you?

Student: Inspection dot passed.

Guest Instructor:Inspection dot passed, right? Now you want to read into passed whatever the manager passed in inspection dot passed, okay. So you want to set passed equals to inspection dot passed. Now you know that if passed now is two then you're not gonna do this anymore and you're gonna move on, okay. You're done with this point but remember that you were locking the manager's office, right. You no longer need to lock the manager's office. Make sense?

So you want to signal – so you waited on the lock in here. You want to say that it semaphore signal inspection dot lock. There is something missing here. I will finish it when we talk about the customer.

Now back to your question. So the question is the following, why do we need to have lock and requested. Why do we need those two? If lock already grants us access to the manager, right? So think about what would happen if you don't have them, okay.

So let's say you don't have the lock. It's obvious that you need it, right? Because you will have two clerks to go to the manager at the same time, right? Now, think about if you don't have requested. Then what will the manager do? Not at all, like, basically the manager will not wait for any inspection, right? It will go on and proceed with passed, give it a random value and go on and keep going in the loop, right?

Now which clerk will be checking its own inspection dot pass you don't know, right? This lock is crucial and it's crucial that you maintain this – this is maintaining this kind of critical section in here, right?

This lock is extremely crucial because basically you don't want to release this lock until you read the value of inspection dot pass. Note that this value is shared among all the clerks and the manager, right? It's a single value. And because we allow only a single clerk to be with the manager at the time then we can safely assume that the value in here will be up by this clerk that is with the manager.

But you have to insure that no one else can be in that portion, okay. You have to insure. So this lock insures basically that you have this critical section in here. It insures that you can read this passed before you release the lock and before any other clerk can be with the manager.

Because potentially let's think about it differently. What happens if you switch those two lines? What happens if we first signal the lock, basically release the lock and then check the value of what's in inspection? What's in passed? I'm sorry.

Student:[Inaudible] inspection dot passed.

Guest Instructor:Potentially, you're not sure, potentially. What happens is if we had those two lines kind of flipped what would happen is probably you could have the inspection dot lock signaled, which means that the lock of the officer of the manager is no longer locked and any other clerk could get in.

Now if you're Fred, which is this clerk, Fred, gets pulled out of the processor at this point it is very possible that some other clerk could get into the managers office, show him his cone, get his passed value of right original pass value before this guy gets back the processor. Do you understand that? All right?

Now another thing, let's see, what would happen if the manager was the one to signal an inspection lock? Like he would say, "Oh, here, let's give the control to the manager." And the manager has his office lock and he could just unlock it. What is the problem with basically moving this line from here to the manager? You see any problem with that?

Student: Isn't it the same person.

Guest Instructor:It's exactly the same thing. It's basically the manager is gonna unlock the inspection lock. So he's gonna unlock his office and we don't know if this guy actually did execute this line or not, right? So, again, the lock would be opened, the door would be opened, any other clerk could get in, could override the passed value before this guy gets to here. Make sense? Yes?

Student: What is – so the manager waits on inspection dot requested. What if the manager just waited on the door being unlocked? Because it seems like once you unlock the door you make your request at the same time. It's like the same thing.

Guest Instructor:So what you want to do is the following, you want to have the manager waits on the lock.

Student: Yeah.

Guest Instructor:And you also have the clerk wait on the lock.

Student: The –

Guest Instructor:You see the problem with that?

Student:– clerk waits on finished. The clerk doesn't even –

Guest Instructor:What would happen is you have a manager waiting on lock. You have two clerks signaling lock and getting in. You see the problem? The thing is binary rendezvous it's always you need those two things because one is waiting for the other to finish something and the other is waiting also. It's exactly the same idea of the Buffered Writer and Reader which is also generally called the Consumer [inaudible] Problem.

So you have some consumer that consumes some [inaudible] of something, another consumer that consumes the same thing. Now this is exactly the Buffered Reader and Writer. And what happens is somebody's waiting for the other to produce and the other is waiting for him to consume, okay, because there is some shared source in the middle. So you always need those two things to maintain basically the two – you see, the two basic synchronization of the two agents. Whoever is producing waits for the other to consume and the other way around. Make sense? Any other questions?

All right, so we're basically done with the manager and part of the clerk. We'll have to revisit the clerk when we talk about the customer. But before we move to the customer then now we're gonna be talking about the relation between those two guys. And we'll also talk about this guy, the relation between those two.

Let's first start with the easy part. Let's see where it's put here. So we have a void customer and the customer takes the random number of cones that he needs to order, right? So he takes an int num cones and let's see what he does.

The customer gets in the store he is browsing. This is just wasting processing time. Nothing, right? He's gonna spawn N num cones clerks to do every single cone for him, right? So he will basically need to get num cones clerks, like if this is three clerks to work on his cones and wait until they're done. Okay, before he can move on to the cashier. Make sense?

So this means that we will need to have some semaphore and why we need to have it because we know that we need to wait for all the clerks, okay? So we need the semaphore. Let's call it clerks done. And this one we initialized to zero again, okay.

And then for int I is equal to 0, I is less than num cones, int I plus plus, what you want to do is basically fire each clerk for your cone. So you do a thread new right at the end, and then you fire the clerk. That's a debug name. The function is the clerk, right? And now we need to pass to the clerk the semaphore because you will wait on the semaphore for all the clerks and you want every clerk when he finishes the cone to signal this back to you which is equivalent to handing – to giving you basically the cone. Make sense?

So you want to thread new a clerk passed to it one parameter which is basically the clerk's done semaphore. Now this takes us back to here, okay. So this clerk takes a semaphore, let's call it sema to signal. And once it's done making the cone, inspecting it, approving it and it's all done then you basically want to semaphore signal the sema to signal. Make sense?

Now that's for the clerk. Now we're done with the clerk. This guy just spawned num cones clerks, right? He needs to wait for all of them. So he saw last lecture how we do that. You just have another for loop for int I 0, I less than num cones, int I plus plus. You want a semaphore wait on clerks done.

Now we know that this guy will never get past this point until all the clerks are done with all his cones, right? At what point you just could basically go and free this semaphore because you won't use it anymore, okay.

So you could just go here and is it free semaphore or semaphore free? Semaphore free. Okay. You can double check that. Semaphore free, clerk's done. And then you walk to the cashier. Starting at this point you will see how the synchronization works between a customer and the cashier and I think this is the most tricky and the hardest part of the whole thing. Yes?

Student:[Inaudible] the semaphore wait on the for loop [inaudible]?

Guest Instructor:That's a very good question. So the question is the following, why don't we include the semaphore wait call and the for loop in here to just have a single for loop that has basically the thread new and the semaphore wait. Anybody has an idea? Yeah, go ahead.

Student: Because we'll – the clerks will only be spotting once at a time?

Guest Instructor:Yeah, any other ideas? Anybody [inaudible] that? So what happens if you do that is the following, you get a clerk and you wait on done, right? Now you did actually initialize clerk's done to zero, right, in semaphore new. Once you wait on this you basically block, okay.

So you will be blocked in this line. You want be able to go any further in the for loop. You understand what I'm saying? Until this guy who you just spawned does a semaphore signal on done so you can get to the next one. So basically you'd be doing it one by one, okay? Yes?

Student: Can you make this [inaudible]?

Guest Instructor:Right. So the question is the following, can we make the semaphore and a shared integer value variable or a [inaudible] to variable. And so this takes us back to the problem of [inaudible] end use, okay.

So the main reason why we using semaphores or locks or those basically concurrency constructs is that we need to insure atomicity. And by atomicity, we mean that we want to decrement the value or increment it and check its value in one operation or in several operations that are guaranteed to be done atomically. So if you have an integer, the thing is you will increment or decrement it and then you will check its value and between the two you could get of the processor, okay. Yes?

Student: Yeah, how does it know what threads it owns? Like, if – say you've done the for loop and you went out of bounds by one, so you did like num cones plus 1. How does it know what threads to wait on like –

Guest Instructor:How does it know what threads?

Student: Well, like you say semaphore wait, clerk's done.

Guest Instructor:Yes.

Student: And you're waiting on the semaphore but, I mean, you've created that semaphore four times already for the four different threads.

Guest Instructor:That's correct. You don't really care to know which thread is gonna signal it. You don't care to know or at least in the – in terms of the function itself you don't care to know which one is gonna signal you as long as you know that somebody's gonna signal you.

Student: So probably one of the threads you're waiting for on semaphore wait could be some other customer's thread that was a clerk's done?

Guest Instructor:One of the ones in – no. Know that I am actually having this as a local semaphore in here.

Student: Okay.

Guest Instructor:So if any other customer has also another local clerk's done that's a different issue. It's a different semaphore basically. All right? Any other questions?

All right, so you recently walked to cashier and from that point on we'll be dealing with the cashier and the customers. Now, I was just saying that this is a tricky spot and this is the tricky spot because of the constraint that we impose which is to handle all of them in a FIFO order. Basically whoever goes in line has to be first, okay.

And now you have to think about what goes in the second struct. So, again, we're having a struct in here basically because it makes sense that we group all the semaphores that have to do with the inspection and the clerks and the manager together, right? But there was nothing that would have – kind of prevented us from having them all independent, just having grouped the semaphores up there. We're just grouping them just so it makes sense to you, okay.

So let's look at the other struct. So we have another struct, we call it line. And this struct has all the semaphores that would handle the communication between the customer and the cashier.

Now the customer has to go to the cashier, it has to stand in line and it has to wait for its turn. So basically it has to take a number, right, which is the next place available in the line, okay? So that's have in here something called the number. And this number – this struct again with all the fees that I'm gonna put in here is supposed to be initialized in semaphore and setup semaphores in here, okay, in the very beginning. I just kept it to the end so that it makes sense with what we're talking about.

So the number would be initialized in the beginning to zero because this is the first place in the line, right? Nobody's there yet, okay? Now the cashier is gonna be waiting until somebody shows up in line, right? And when somebody shows up in line this means that a cashier is requested, okay.

So we have a semaphore that the cashier is gonna wait on saying, "I'm waiting until I'm requested." Okay? Until somebody basically shows in line. So we have a semaphore requested and this is going to be initialized to zero.

We also have another semaphore; this is a little tricky, now each customer with the cashier they have some kind of synchronization going on, right? Every customer has to go and say, "Okay, signal me when you are done with my bill. And it has to be me because I am waiting in the queue in this specific position." Right?

So there should be some – it's not really that we are the customers and we are waiting for you, cashier, to signal somebody of us. It's not really somebody, it's not any one of us and somebody will leave. We're not caring about all the customers leaving in the end; we are really caring about every single one of them leaving in his turn. Right?

And this should give you an idea about having this kind of struct that would have a semaphore that is a rendezvous semaphore between every single customer and the cashier but this semaphore has to be for every single customer because it has to maintain the order of every single customer. Does that make sense?

So in here we will have a semaphore, let's call it customers, and those are the customers that basically requested the cashier's service but are waiting to be serviced, okay? So this is gonna be an array of ten semaphores, one for every single customer in turn, okay. So you come, you pick a number, you know you're number three, then you are gonna wait on the semaphore of three. And the cashier, once he gets to your turn he's gonna signal customer of three so that's you – it's your turn to leave. Make sense?

Student: Why ten customers?

Guest Instructor:Because we just assume ten customers. You could have a single number, yeah.

Student: So one point that the customer in the front of the line makes their request and that – and then he leaves and then the next person makes their request and he leaves so that the line of clerk only deals with the one – only one customer.

Guest Instructor:This is basically what we're doing. What you're trying to say is how can we – so let me ask you the question, okay? How can you insure that the customer – the first one in the line is the one that came in order? Because basically remember that you are not keeping track of the state. There are customers that finish and they get all their cones and they go to the cashier, right?

You don't remember who came first. You don't remember the order. You don't remember which one picked the number, right? If you say the one in the beginning of the line that's what we're doing basically, we're keeping track of who is at the beginning of the line.

We’re gonna say if it’s the one in place number zero and move on to one which is basically the beginning of the line. If you say that they shift, right? But if you don’t do that then you don’t know because you have like threads that finished at random times and you have no clue whatsoever which finished first. Make sense? Yes?

Student: If you had multiple semaphore waits and you do just one signal will all of them wake up or just one wake up?

Guest Instructor:Are you talking about those arrays that I’m having here? When you do a wake up you wake up the specific semaphore. You don’t wake up any one of them.

Student: But generally do you just [inaudible] waiting on a semaphore?

Guest Instructor:At least if you do wake up –

Student: At the signal.

Guest Instructor:– what happens typically – this is an OS issue, but typically what happens is when you have somebody waiting on a semaphore it’s put on a block list, right? And so you have all those threads that will be put on the block list and it depends on the scheduling issue. So it depends on whether you pick a shortest job first or you pick a priority scaling, whatever. But whoever gets to be scheduled once he finds that his semaphore was signaled then he will be able to proceed, okay?

All right. So now we have the customers and we’re still missing one thing which is very similar to the one thing that we missed before. Can anybody see it this time?

Student: Having a lock on the cashier?

Guest Instructor:Having a lock on what?

Student: On the cashier.

Guest Instructor:Having a lock on the cashier. Why do we need the lock?

Student: Because they can only deal with one customer at once.

Guest Instructor:But actually you can see that the cashier is maintaining this thing because the cashier is the one that’s gonna be waiting for requests and he’s gonna be spawning basically – he’s gonna be signaling one of those. Anybody sees any other problem?

Student: Yes, a lock for the number.

Guest Instructor:Exactly. We need a lock for the number. Basically you can think about this. Two guys go, they check what is the number. The number is zero. The other one also – and the first one gets out of the processor. The other one gets the – gets to be scheduled, checks the number, it's still zero and they both increment the number; they both get the number one, right?

So this is why, again, we need the lock on this number to make sure that this is the shared resource, this is where we have race conditions and this is what we want to secure, okay? So let me have a semaphore lock and to what is this semaphore gonna be initialized?

Student: One [inaudible]?

Guest Instructor:Yes, it's gonna be initialized to one and I think I will need to write the last few lines very quickly because we're running out of time. All right, so let me write the cashier first and then we go to this guy.

Here's what the cashier's doing, very easy. This is the cashier. It's not taking anything. And what the cashier's doing is basically for the number of customers what he's doing is he's waiting to be requested. Once he's requested he handles the bill, checks out the customer and then signals the semaphore, okay.

So you have a for loop for int I is equal to 0, I less than 10 int I plus, plus. What you want to do is semaphore wait on line. So this is line. So you wait on line until requested. Once you are requested then you know that you are basically servicing the first customer in line. So you check out some function. Check out customer I and then you signal customer I because you know he's the first one in the queue – in the line.

So you semaphore signal line dot customer of I, all right. And that's your cashier. Now let's get back to the customer. So you walk to the cashier, okay? What you want to do is you want to signal to the cashier that you request a service and wait for the cashier to handle your bill, right?

So you want to semaphore signal line dot requested and then you want to semaphore wait. Oh, I forgot something very important. I don't know my number, right? I didn't even get a number, okay? So before you request you want to get a number but before you get a number you have to lock, right, because you don't want anybody else to be getting a number at the same time.

So we want to semaphore wait on line but lock. You want to get the number so we simply an int, let's call it to replace, this is equal to line. – what did I call it? Number plus plus so that next time is the next place on the line, okay?

And then you want to signal definitely this lock because you are done using the number. And after you signal this lock, maybe right here, you want to signal to the cashier that you are requesting his service. So you signal line dot requested and then you want to wait until he handles your bill.

So basically wait on line dot customers of your place, okay? And this is your customer. I don't think I'm missing anything. Does it make sense? Are you guys confused? This is hard. This is not an easy – it's not that it's – it's not as complicated, it's just that it's just a lot of problems going on, a lot of synchronization issues going on.

So I would definitely suggest that you guys go and do this again, okay? You have to think about it. You have to think about the interactions between all of the players. You have to think why you need those semaphores and how you're gonna use them. I'm gonna see you again tomorrow in this section. We're gonna go over another example for the studying and it's gonna be related to your assignments – to your project, okay?

Okay, have a good day.

[End of Audio]

Duration: 52 minutes

ProgrammingParadigms-Lecture19

Instructor (Jerry Cain):Hey, everyone. Welcome. I have a glorious set of handouts for you today. I have a midterm solution and I have the first two handouts and a stream of a few handouts I'm going to be giving you on our next paradigm and our next programming language. You'll see that I do have the midterm solution in there. I actually distribute that, obviously, so you can check your answers, but also so that we're somewhat transparent in how we actually grade these things.

Turns out that these things amount to like 25 percent of your grade, so I like you to know what criteria we're using to consistently grade everybody so that if you see something that is so clearly wrong in terms of the way we graded it, you can confirm that by looking at the grading criteria and then coming to me if you have problems.

Usually when there's a one-point error that's not listed specifically on a criteria, that doesn't mean I'm gonna give it back to you. It means that you made a mistake that we didn't anticipate anybody making and we didn't put it on the criteria. But if you have a total of five points taken off for a problem out of ten, and it's not clear why you have that many points taken off, that's a difference scenario, and so consult the answer key, and if there really is disparity, come and talk to me about it.

You should come and talk to me about it if you're worried about there being a discrepancy because this thing does just end up counting a lot. So don't feel shy about coming back and asking for clarity as to why we took points off that we did.

So what I want to do today is I want to introduce a new programming language, and I want to first illustrate a new paradigm, one that you've certainly not seen before unless you've coded in other classes at Stanford or coded prior to Stanford. We have spent a lot of time talking about these two paradigms, imperative. You'll also hear me call it procedural.

They're not necessarily the same paradigm, but the language we're using to illustrate both of them is the same. We focus on C as the representative of those two paradigms. We also have the object oriented paradigm UC++, and even though we know that CSC++ ultimately translate to the same type of assembly language that you kind of think about the problem differently or we think about our solution to the problem differently when we take a pure C approach versus a pure C++ approach.

The reason this is called imperative or procedural is that they're focused around the verbs that come up in the paragraph description of a solution. Think about what a main function looks like in a typical C program. You declare all your data, and then you make a series of call to these very high level functions like initialize, do it, terminate, and then return zero or something like that. Do you understand what I mean when I say that?

Okay, so the first thing you see associated with any C statement is usually or very often the name of a function that gets invoked to kind of take control of 10 percent of the program or some small percent of the program just to get something done. As far as object oriented is concerned, you're used to something like this. I'll write the C equivalent in a second, but C++, over here you declare vector V.

You do something like vector new where you pass in the data and a few other parameters. You do things like vector insert ampersand of V, and vector sort of ampersand of V with additional arguments. Those aren't prototypes. That just means I don't feel like spelling out the rest of the call.

In C++, you declare a vector maybe of Nths called V, and you do something like V dot push back of 4 or maybe you do something like V dot erase of V dot begin to remove the front element. I know you haven't dealt with that specifically. Don't worry about the fact that you haven't necessarily used all those methods, but clearly, in this exactly right here, you're looking at the verbs first, okay.

It is oriented around the procedure, so I'll just go ahead and say that it is procedure oriented whereas right here you declare this object up front. And the first thing that comes up in a statement of any particular – comes up in any one particular statement, usually the first thing you look at is the piece of data that's being manipulated.

In this case, V is the data. It's also called an object. Because this is up front, it looks like each statement is oriented around the object, which is why it's called object oriented as opposed to procedurally oriented, okay. Does that make sense?

So you may think, "Well, I don't understand how I could possibly program it in any different manner." Well, even sequential versus concurrent programming, there's a little bit of a paradigm shift. You have to think a little bit differently about the problems that are gonna come up when you program in a threading environment versus a non-threading environment, okay.

Usually when – this is a little bit of a caricature, but this is really how I feel. Whenever you're coding up a normal program like programs one through four, you have this very linear way of thinking. You have a series of tasks you want to get through, and it's almost like you're inside the computer like typing things out one by one. But when you're programming in Assignment 6, it's at least not all of it has to do with execution logic. A lot of the hard stuff is like figuring out how all the threads are gonna interact.

And so you're thinking about multiple things at a time. And I'm actually like standing up a little bit more because I actually think with the back of my head when I'm programming concurrently because I'm trying to imagine all of these different scenarios of thread interaction that I have to worry about that have nothing to do with code, right? I actually have to see all these little players in like some thought cloud and how they might be interacting and how race conditions might come into being.

And so concurrent programming and multithreading is it's own paradigm that isn't really tied to any one particular language. Object orientation isn't tied to C++ any more than it's tied to Java or Python or any other LON, which you might know. And even though C is probably the only procedural language you've really dealt with, there's Fortran, there's Pascal. Those things really do exist, not because a lot of people are writing new code in them, but there are Legacy systems from 20 years ago that still exist, and even if they're not adding features to that code base, they're certainly maintaining it and fixing bugs that crop up, things like that.

The amount of energy that was invested in fixing COBOL code bases back in like the final three months of 1999 was outrageous because everyone was totally petrified of the Y2K threat, that because we weren't storing years with enough information that everything was gonna go back and jump back to like year zero or 1900 or however they actually started it.

It turned out to not be nearly as big of a problem as they thought it was gonna be, but everybody was working in a procedural language called COBOL for a good amount of 1999, not everybody, but a good number of companies were, okay.

What I want to do now is I want to stop talking about procedural and object oriented for a while and go back to sequential programming for the most part and start talking about what the functional paradigm is.

Now functional and procedural sound similar, but procedure, if you're a purist about the definition, it is a block of code that gets called where you're not concerned about a return value, okay. Does that make sense to people?

Like you have to think about a procedure as a function that has void as a return value. When I talk about functional, I'm talking about procedures and functions again, but I really am oriented around the return value, okay.

We're gonna study a language, I think it's a very fun language to learn, called Scheme. There are aspects of Scheme that are interesting. I want you to invest a little bit more energy in understanding the paradigm than the language because the paradigm is – features of the paradigm are interesting takeaway points from a class like this if you're not gonna program in Scheme again, which probably will be 90 percent of you.

But nonetheless, the functional paradigm is very much oriented around the return value of functions. So let me just do an aside right here and just think of pure algebra, nothing sophisticated. But if you had a function like this right there, don't even think about code. Just think mathematical function. That looks like – it's the name of some function that takes two real numbers or two complex numbers or whatever, two pieces of data, and in traditional math, you know that it just returns or evaluates to something, okay.

So it may be the case that in a mathematical setting it's X cubed plus Y squared plus 7, okay. And in a pure math setting, you know exactly how to evaluate that if it stops being parameterized on X and Y, and you actually pass in 5 and 11, okay.

If I do something like this, then you know exactly what I mean when I write that down, okay. It turns out that the definition of G of X involves the definition of X where it takes this one parameter and splits it out into two parameters so it can call F. And then it incidentally adds eight to whatever is returned there. Does that make sense? Okay. And if I go so far as to do this, maybe it's the case that H of X, Y, and Z is actually equal to F of X and Z times G of X plus Y, and that's just the associations that are in place, okay.

Now this isn't legal code. This is just math. What the functional paradigm approach does is it assumes you have lots of little helper functions that are interested in synthesizing one large result. So maybe it's the case that I'm interested in the result of H where it gets a one, two, and a three.

And I happen to decompose it this way. I could actually inline the definitions of all of those things, and I could frame H in terms of just X and Y and Z, and not have any helper function calls whatsoever, right. But for reasons of decomposition, it's usually easier to frame things in terms of helper functions, and that's kind of what I'm doing right here.

What Scheme and what the functional paradigm tries to emphasize is that you give a collection of data to the master function that's supposed to do everything for you. It does whatever is needed in place to synthesize the results, and that answer is returned via the return value, and that's all you're interested in, okay.

Maybe it's the case when this is been one, one, and four. I have no idea what the numbers are. Maybe it returns 96. I have no idea. And I'm only interested in the 96 because that is the material product that I'm trying to get out of the program.

What a functional paradigm approach would take is that it would just say, and it would associate something like, how do I want to say this? It would do this, and it would associate it with F of XZ times G of X plus Y, or it might actually prefer to write it this way, is equal to X cubed plus Y squared plus 7 times F of X plus Y, X plus Y plus 1. And that's plus 8, okay.

But you would actually write it this way and really expect F and G as functions to themselves return values that contribute to the synthesis of a larger result. Does that make sense to people? Okay, question in the back.

Student: Why isn't the Y squared replaced with Z?

Instructor (Jerry Cain):I just – I didn't go that far. Where did I do? I'm sorry. I just messed up.

Student: Yeah.

Instructor (Jerry Cain):Yeah, sorry. Thanks. So rather than actually trying to do this in terms of pure math, let me just give you an example of what a Scheme function looks like. I'm not even gonna try and explain what the syntax is. You're just gonna have to intuit what it probably does as a function.

I'll get to the pure syntax later on, but you can kind of gander what that as a keyword is probably gonna do. And I'm just going to do this. Let's say Celsius to Fahrenheit. It takes the temp, okay. And I just do this. I'm going from something at temperature, so what I want to do is I want to multiply it by 1.8 and add 32. This is how you would do this, okay.

Now you're not sure what the syntax is, you can kind of see that the right numbers come up in the conversion of Celsius to Fahrenheit, okay. So I want to scale 0 degrees or 100 degree by 1.8, okay, and then actually add 32 to it. And that's how I got 32 or 212 out of it.

So I'll go over syntax later, but what's really happening here is that in a Scheme environment, which is an example of a functional language, it associates this is a symbol, and the actual dash and the greater than sign forming an arrow, that's actually a legal part of a token in Scheme. They want you to be as expressive as you could possibly be using the full alphabetic or a full [inaudible] set, pretty much the full [inaudible] set to name all of your symbols.

It's framed in terms of this one parameter, and as a function call, it's equated with this expression right here where whatever value of tenth is supplied replaces that right there. So if I go ahead and I type this in to the shell, to the actual Scheme environment, it's supposed to somehow pop out a 212, and it succeeds in doing that because it takes this as a recipe, stops – there's a template on the tenth variable, actually figures out what it would evaluate to if ten became a value – came down to 100.

As an expression, it evaluates the 212. And so this as an expression is equated with this expression, and it comes back with a 212, okay. Does that make sense to people? Okay. Don't worry about the mechanics. Just think about the actual description of what a functional language is trying to do here.

Now let me actually just describe what the Scheme environment is like. We're using an open source product that I happen to – well, I didn't work on it, but I used it quite a bit a few years ago for a consulting job. It is a product called Kawaa. And I don't want to say that it's standard, but it happens to work fairly well and I just wanted to use it and I did because it's free and it's open source and I can just install it in 107 space, and then nobody has to – I don't have to bother anybody in trying to get support for it.

When you launch this thing called Kawaa, you more or less launch an environment that functions much like the shell where you type LS and make – and CD and things like that. It just happens to not speak the batch language or the TSCH language or TCSH language. It's actually a little bit more elaborate that something like Bash or SH or something like that where you actually go ahead and you get a prompt, and it always expects you to type in something that can be evaluated.

Very often – not very often, but it can be the case that you type in things that are very, very simple to evaluate. If you type in the number four, then in its little functional way, it says, “I have to let that evaluate to something.” It’s gonna do it. It’s gonna have a very easy time doing it, and it’s gonna come back with A4, okay.

If you go ahead and you type in the string hello, then it, itself is also considered to be atomic strings, or more or less atomic types in Scheme, or at least we can just pretend that they are. So it will print out hello because that’s what the hello string evaluates to.

If I want to deal with Booleans, it turns out that pound F is the Boolean constant for false. It’ll actually print this out for you. If I want to print out true, I can do that. It’ll print out true. If I want to deal with floating point numbers, I can continue up here. You may think that it’s going to be very clever about things like this, but if I type in 11/5. That looks like it’s a request to do division.

It’s not. You happen to type in a number in the rational number domain, and so what it’s gonna come back is oh, that’s just 11/5. Thanks for typing that in, okay.

If you go ahead and you type in 22 over 4, it will go ahead and reduce it for you, okay. But it usually stays with – it preserves type information as much as possible in going from the original expression to whatever it evaluates to, okay. Does that make sense? Okay.

The one composite data structure that is more or less central to Scheme programming, at least how we learn it, is the list. There are a couple of things that can be said about the list, but let me just put a list up on the board. If I do this, then technically, what I’m doing is I’m typing in a list. It happens to be framed in such a way that I ask it as a list to invoke the plus function against all of the arguments that follow it, okay. Does that make sense?

So the list is the central data structure in List and Scheme. We happen to be dealing with a dialect of Lisp called Scheme. Lisp would be a better name because that’s short for List processing, but we’re having to use an earlier version of Lisp called Scheme that was invented by John McCarthy, who’s at Stanford now, but like some 50 years ago when he was at MIT as a – just as a untenured faculty professor at the time.

He just wanted to illustrate how closely tied mathematics and programming languages can be made to be by coming up with a programmatic implementation of something called the lambda calculus, which is basically some very fancy phrase for coming up with a theory on functions and how they evaluate, and not necessarily restricting them to real numbers and fractions and things like that, to let functions arbitrarily deal with floating points and Booleans and strings and characters and lists and hashes and things like that, okay.

If I – obviously this would put another six. If I do this times plus four, four, plus five, five, I do that right there, I’m dealing with nested lists where what were previously housed by simple [inaudible] before are now – those possessions are now occupied by things that are themselves lists that should be recursively evaluated, okay.

So whereas this evaluated to one and a two and a three much like this evaluated to a four and a hello and a false constant. These evaluate to themselves, and then they participate in the larger function evaluation that uses plus to kind of guide the computation, okay. And not surprising, you'd get a six there. That four would evaluate to a four. That four would evaluate to a four. This as an expression would evaluate to an eight.

Five would evaluate to a five. Five would evaluate to a five. This entire thing would evaluate to a ten. The overall thing would evaluate to an 80, and that's how you get an 80. It's not the math that's interesting. It's actually the manner in which things get done, I think that's the most informative here, okay.

So I'm willing to buy that even these are function calls, I don't like using the word function call, okay. I mean I think function call is fine. I just don't like to speak of the return value of a function call because that's a very imperative procedural way of thinking about it. I like to think of this as evaluating to a 6 or an 80. Does that make sense to people? Okay. Okay.

So there's that. It turns out that plus and asterisk are built-ins. All the mathematical operators you expect to be built-ins are, in fact, built-ins. If I'm curious as to whether or not the number four is greater than the number two, I can ask. Isn't the case that four and two ordered that way actually to respect that greater than sign as a function. They still didn't return a number of a string. It returns a Boolean.

This would come back with something like that right there, okay. If I did this, less than – is ten less than five? That would come back with a false, okay. If I – this is just the prompt. That doesn't agree with that sign.

If I did something like this, it would assemble things in the way you'd expect, okay. It actually even does short circuit evaluation. So this one right here would be evaluated. This four as an expression evaluates to a four. This two evaluates to a two. This over all thing evaluates to a two. This ten evaluates to a ten. Five evaluates to a five. This overall thing fails, and it's at this point that the conjunction that you would just expect to be in place with ends would overall evaluate to a false, okay.

Now I am assuming you're gleaning the fact that the zero parameter or the zero position in a list the way I'm using them right here always identifies some form of functionality. There's functionality associated with this symbol right here, okay, and it knows how to take these two arguments and produce other true or false.

The same thing can be said right there and right there, okay. But the actual symbols that are attached Old Testament functions always occupy the zero place. It has this very prefix oriented way of dealing with function calls, okay. Does that make sense?

One of the most complicated things about Assignment 7, no joke, is actually getting the parenthesis right. You're so used to typing it at the end of a function, and then typing an open paren after it that that's what you'll type out, and then there's so many parenthesis surrounding you anyway when you're typing this stuff up that it's very easy to miss it.

So you have to be very – and just balancing the parenthesis isn't enough. You have to make sure that you get into this habit of just opening up a parenthesis, thinking like you have this entire list of things that help express some kind of function call, and just know that that's the type of thing that's really hard to get right when you write your very first function in Scheme, okay.

Now there is – there are a couple of things about lists that I want to go over before I start defining my own functions. I told you that Lisp, even though we're doing it with Scheme, we're really doing it with Lisp. It's called List processing for a reason. Everything, including function calls, come in list form. The only exceptions are things like for's and hello's and things like that, the atoms of the data types.

But normally anything interesting is bundled in a list. We don't really have – they do have structs in Scheme. They do have classes in our version of Scheme. We're gonna pretend like those just don't exist. Everything that's an aggregate data type is just gonna be packaged as a list. And we're gonna know that the 0th item in the list stores like the name. And the 4th – I'm sorry, the 1th slot in the list stores the GPA or the address or the phone number or something like that, okay.

What I want to do is I want to go over a few fundamental operations that are technically functions in Scheme that allow you to dissect and build up new lists. You're not gonna always want to return a 212 or a hello or an 80. A lot of times you're gonna want to return a list of information, or a list of lists, or a list of list of lists, or whatever happens to be needed in order to present the overall result to you.

There is a function called CAR. There's another one called CDUR, and there's one calls CONS. I'll go over why they're called this is a second. It's not really important. It's sort of interesting, but then it stops becoming interesting after a few seconds.

Car and CDUR are basically list dissectors, okay. If at the prompt, I type in CAR, I'll explain what the quote is in a second, one, two, three, four, five. This returns the number one, okay. If you just think about these things as link lists, they kind of are link lists behind the scenes. Car is associated with the data that's housed in the very first node, okay. It always is the evaluation of whatever is occupying the zero slot of a list, which is why this one comes back, okay.

If I ask for the CDUR of the very same list, it basically covers everything that the CAR does not, so it returns the rest, which in this case would be two, three, four, five, okay. Does that make sense to people?

If I do this and I nest them, I've asked for the CAR of the CDUR of the CDUR of one, two, three, four, five. It does recursive, that evaluation, in this bottom up strategy, comes here and identifies two, three, four five as a list, three, four, five as a list, and then the CAR of that is the first element of what was produced by this, which was produced by this. So this would come back with K3, okay. Does that make sense?

Now what's this quote all about? If these quotes weren't here, Scheme, and it may seem weird to you at the moment, but this is actually a much simpler language than CSC++, and I'll have several defenses of that in just two minutes. But if I take this quote away, then this right here is supposed to be treated just like this and this right here. Do you understand what I mean when I say that?

So it would actually look for a function called one, okay. And when it doesn't find it, it's gonna be like, "Whoa, I can't apply the one function to two, three, four, and five." So it would issue an error right there, okay. In our Kawaa environment, it'll throw a Java exception to advertise the fact that it's implemented in Java, but nonetheless, it will break, okay.

So you don't want to take the CAR of something that doesn't actually evaluate by putting the quote right there. It just is an instruction to suppress evaluation, that the list that is being presented after that quote is really raw data that should just be taken verbatim without any kind of recursive evaluation, okay.

It's actually shorthand. Whenever you see something like "One, two," and I could even do this like that right there, the quote just says basically to the parser, "Stop evaluating." From everything from the parenthesis that I'm looking at to the parenthesis that matches it, okay. It's technically shorthand for this right here.

It matches that, matches that, matches that, matches that, and quote is just this metafunction in place that doesn't actually – it kind of evaluates its arguments, but as part of the recipe for this quote function, it just doesn't evaluate its arguments. It just takes them verbatim, okay. Does that make sense to people?

You're gonna type it this way. You're not gonna use the quote function. There are all kinds of nifty variations on the straight, flat quote. I might go over it in a section handout, but it's so ridiculous. There are actually variations on this where you can actually use the back quote and the forward quote and the comma, which are variations of this right here, to suppress evaluation temporarily and turn it back on internally, okay.

But I just want to have this one thing where everything recursively is not evaluated, okay, and not deal with these variations. You can read about them if you want to, but you won't have to use them for anything that we do in this class.

Okay, so that is the way to suppress evaluation. That's gonna be very good because if we're gonna want to express all of our data in list form, we don't want to be penalized because we're using Lisp, that we always have to have some function evaluated in our data, okay. We might just want to present our data as these bland lists, okay, and package them in a way that we just deal with consistently, okay.

So CAR is like synonymous with first. In fact, some dialects of Lisp actually first defined as a function. Coulder is synonymous with rest. It's like whatever you get by doing – following the next pointer behind the scenes, okay, whatever list you arrive at after the first element.

And you can use these in any clever way you want to to get to the third element or the last element or this element right here expresses a list, okay. Whatever you need to do to package – to get to your answers. You can package CAR and CDUR in some – whatever clever way you want to, okay.

Now why are they called CAR and CDUR? It's really not a very interesting reason, but they – I mean it's kind of interesting. There was a – the original implementation of either Scheme or Lisp, I'm not sure, was just on an architecture that had two exposed registers. One was called the address register, emphasis on the A, and one was called the data register.

And the CAR and the CDUR that were most recently dealt with the addresses that implemented them were stored in the address register and the data register, okay, and that's where the AR and the DUR come from, address register and data register. Does that make sense? I don't know where the C came from, something related to the letter C, I'm sure. I just don't know, okay.

So that's why they're there. Our system doesn't define any synonyms to these. Some versions of the language define first, second, third, fourth, all the way up to tenth I've seek, okay. But ours doesn't, so you really have to deal with the raw CAR and CDUR calls, okay.

These two functions take lists and break them down into their constituent parts. CONS is kind of the opposite. If I, at the prompt, do this, CONS, and I say one, which evaluates to itself on the list two, three, four, five, CONS is short for construct. It actually synthesizes a new list for you, and it would return this. So CONS is always supposed to be – take two arguments.

The first argument can pretty much be anything. The second one is supposed to be a list because what more or less happens is that it takes this element right here. It pulls this – it effectively pulls this parenthesis in like it's on a spring or something and drops the one in front. And whatever you get as a result of that is your resulting list. Does that make sense to people? Yes, no? Yeah.

Student:[Inaudible] two, three, then [inaudible] four five? So you can put one in between three and four?

Instructor (Jerry Cain):You could, but you – I'm sorry. So tell me what you want me to write. You want a CONS call right up front? And then what?

Student:[Inaudible].

Instructor (Jerry Cain):Write it like that?

Student: Yes. [Inaudible].

Instructor (Jerry Cain):Four, five?

Student: Yeah.

Instructor (Jerry Cain):Yeah. I mean they're – actually I know what you're trying to do now. That would not work. CONS really has to have two arguments, and the second one has to be a list, okay. If you wanted to do – let me just – in two minutes, I'll revisit this example and at least just show you the code as to how you would assemble this.

What I do want to emphasize – let me erase this since it is syntactically a little off. I want to emphasize the fact that it's very literal about how it takes the first piece of data and puts it into the front of the list. If I do this, CONS of one, two, three, and I try to CONS it onto the front before five, I actually will get from that another list where four, five is its CDUR, okay, but I will get this out, okay.

It's very literal in the fact that it takes this one element, which happens to be a list one, two, three, and it kind of prepends it to the front of everything that resides in the second list. So this emphasizes a point. I haven't formally said this yet, but lists in Scheme or any dialect of Lisp for that matter, can be heterogeneous, okay.

Right now I've – almost all the lists I've done up to this point except for one of them, I guess I erased it. All of the lists have been homogeneous in that they've all stored integers or they've all stored Booleans or strings or something like that. That isn't a requirement.

So there are a couple of features so far about this that I think are pretty interesting. There's very little type-checking going on, okay. There's a little bit, but there's not nearly as much of a compile time element to Scheme as there is in CSC++, okay. It just lets you type in whatever you type in, and it's only as it evaluates things that if it sees a typed mismatch, because you try to say add a float or a double to a string, that it'll say, "You know what? I can't do that, okay."

But it's actually at run time when it does the required type analysis to figure out whether or not something will work out. Does that make sense? Okay.

As far as what you wanted to do, there is a way to do it. I'll just introduce it because I can introduce a function pretty quickly. If I really wanted the list one, two, three, four, five out of this, I don't have to use CONS. I can use a built-in called append. And that's not CONS. It actually does effectively remove that paren and that paren right there and build one big sequential list out of everything, okay. So that would give me the one, the two, the three, the four, and the five.

Append, unlike CONS, can take an arbitrary number of arguments. You can even take one list if you want to, but if I gave it this, that would return what you'd expect. It would actually figure out how to return one, two, three, four, five, six, seven, eight. And we'll be able to implement our own version of append in a little bit, okay. But it basically just threads everything together. It's like it removes all intervening parenthesis and whatever list is left in place is the return value. Yeah.

Student: Would it work in three [inaudible] list?

Instructor (Jerry Cain):It would not, which is the next example because that's what – the way he fed arguments to the example he was announcing didn't have one of them as a list, so I will fix that problem right now. If I really wanted to put a one in between a two and a three and a four and a five, I could do this, append – let me put the arguments this way. I will just say two, three.

I will write it incorrectly. I'll say one, and then I'll put the list four, five. And let's, of course, pretend that my goal is to get the list two, three, one, four, five out of that. Append doesn't like this. It wants to see parenthesis around all of its data points, okay. You could actually create a little list around the piece of data by calling this other built-in, and then all of the sudden, that just temporarily, or not even temporarily, wraps parenthesis around it and creates a singleton list so that it can actually participate in an append call, okay.

Now I'm just breezing through all these functions. I will be honest. I've probably talked about half of the functions you're gonna need to learn for the Scheme segment of the course, okay, and none of them really are that surprising. Like list and append, that's not rocket science.

It may be interesting how they work behind the scenes, but it's not like they're obscurely named, okay. CAR and CDUR, yes, they are – and CONS, they are obscurely named, but those are probably the only three that really need to kind of think and remember what they do, but even then that's pretty easy, I think, okay. You guys get the gist of all the mechanics here? Okay. Yep.

Student:[Inaudible].

Instructor (Jerry Cain):[Inaudible] are cool. In fact, they're used a lot. I should emphasize that if you type in something like, let's say, CDUR of the list four right there, what follows the four is nothing, but it still has to be expressed as a list, so this would return that right there, okay. It's fine, and actually, the empty list is kind of the basis point for forming all lists. When I talk about how CONS is implemented, you'll understand that the empty list is kind of like the base case of a recursive call.

I should say that if you do this, that's a no no. Now some implementations will just – whenever you try to take the CDUR of an empty list and try and remove a car that isn't there, that some implementations will just return the empty list for you. I don't want you to program that way.

I want you to assume that either a CAR or a CDUR levied against the empty list is actually an error, okay. And I actually am forgetting right now was Kawaa does because I never try to exploit the feature if it is one. I just assume that this is gonna be [inaudible] in there, okay. So no, it would print no, don't do that, okay. Does that make sense?

So I dealt with every – more or less I've dealt with all data that's been a constant or a list constant or something like that. That's not the way it is. You really do define functions in Scheme as well, or else you wouldn't be able to build scalable systems that can be parameterized in terms of arbitrary datasets.

So I already gave you an example of one function over there, but let me start even a little bit easier. If you go ahead and use the define keyword, define has its own purpose. It's occupying the slot where you normally see pluses or times or divisions, okay. What happens next, if I just do this, add, okay, does that make sense? And I just pass an X and Y, there's no comma separation between the arguments. The space is the delineator.

And I equate this functionality like that, okay. Then you type that in. It actually comes back and says, "Oh, I just defined add. Thank you very much, okay." It actually prints out add, not because it evaluated to add, just because it's the define keyword. It just wanted to remind you what function just got defined.

Now this is the very first example, and this is an obscure point, but I kind of want to revisit this a few times later on. This is the first example of any kind of Scheme expression we've dealt with so far that has some side effect associated with it. And the way you hear that, you may be like, "Well, why is that interesting?"

This purely synthetic approach where it takes the data and it synthesizes a return values so that the overall expression evaluates to it, it does all of that without printing to the screen or updating memory in any way that we know about. You're not passing around raw pointers anywhere. Do you understand what I mean when I say that? Okay.

Even the lists themselves are being synthesized on your behalf. If you were trying to do the equivalent things in CSC++ you would have to declare your list data structures, okay, or worse yet, you'd have to actually define a node type, and you'd actually have to thread together link lists using Malak or New and free or delete or whatever you have to do. You'd have to manage the memory allocation by yourself.

Scheme is so much more of a higher-level language and it's smaller and it tries to do less that it's easier for it to do – take what it does and do it very, very well. The list, as opposed to C or even technically C++, the list is a built-in data structure that's core to the language. So in the same way that we breed string constants and integer constants to C, you can actually express list constants. I don't have any up here. Yeah, I do.

This right here, this knows how to build a data structure to represent the list, one, two, three, four, five behind the scenes, okay. You don't have to manage any of that. In purely functional languages, and we're gonna strive for this in the [inaudible] scheme we're gonna learn, you try to program without side effect, okay.

Only to the extent necessary do you update variables by reference. I've certainly not done any of that yet, okay. I've always just relied on what it evaluated to. Technically, there's a side effect associated with this right here in that in the global name space it associates this add keyword, okay, to be associated with this functionality right here, so that from this point on add, the way I've defined it, it actually is a built-in. It behaves more or less like a built-in just like CONS and CAR and CDUR and list and append all are, okay.

They really are peers. It's almost as if there's a map, a global map of symbols mapping to actual functions, okay, where the functions themselves are expressed as lists, and that map is prepopulated with code for CAR and CDUR and CONS. And then you can add to it programmatically by associating this keyword with this list right here, which knows how to behave like a function. Does that make sense to people?

Okay, so when I do this, add 10 and 7, it comes back with a 17 because it somehow knew how to take this 10 and the 7, crawl this list right here to figure out how to deal with the 10 and the 7 that were passed in, and whatever it evaluates to is what add evaluates to. So it's like you equate this symbol parameterized by these two arguments with this Scheme expression, okay. Yep.

Student: Does case matter? Like why does it give you add?

Instructor (Jerry Cain):That's just – actually, I shouldn't have emphasized that. Case does matter when you're typing these things out yourself. For some reason, and this may not even be the case with Kawaa, I just remember the Scheme interpreter I used in this class forever capitalized everything for reasons that weren't clear to me.

But you should be sensitive to case, but just because I print something out in uppercase doesn't mean anything, okay. I like de-emphasize this. Pretend it's just – don't even worry about it, okay. Yep.

Student: Can you use ellipses and say like add XY and –

Instructor (Jerry Cain):Yeah, you actually – I'll talk about that the last day of the Scheme segment when I talk about these equivalent features to CSC++. You don't do that. You actually use a special parameter that catches everything beyond a certain point into a list.

And when we implement, you'll see a little bit in like two or three lectures what the equivalent of the dot dot dot from CSC++ are. I just don't want to go over it quite yet, okay. I mean I just defined my first function ever and it's add. You can see I'm just not that far yet. Yep.

Student:[Inaudible] can you redefine it later?

Instructor (Jerry Cain):Yeah, absolutely. You can redefine it. Some systems will let you redefine CAR and CDUR if you want to. I'm not recommending it, but if you want to like displace the built-in functionality that's associated with CAR and CDUR and list and append, some implementations will let you, okay. I'm not sure whether Kawaa does or not because I haven't tried, but I just know in the spirit of Scheme and how it's implemented, it's certainly possible to do that, okay.

Now this isn't very interesting. What I will do is I will write a function that just deals with a list as data. Notice that I have not actually typed X and Y here at all. So if I want to do this, there's no problem with the definition itself, but if I try to do this, it's only when it tries to evaluate this expression right here that it says, "Well, I don't like levying a plus against two string constants." And only there will it issue a runtime error. Does that make sense? Okay.

Think about the CSC++ equivalent. You would have had to attach data types to this right here, and you would have had to script this call up in the same file or some other file and compiled it so that at compile time it could detect that this isn't gonna work out. There is really very little compile time element to a Scheme interpreter. It just – when it parses the list you type in, that's technically compilation, but it also evaluates it at the same time.

So there's really very little separation between compile time and run time in Scheme, and because it's an active interpreter system, we just call it the run time, okay. So if I type this in, this would error out.

Okay, so would we all agree that length lists are recursive data structures? Okay, more often than not, if it's linear recursion, you would probably just implement it iteratively. In Scheme, we're gonna take this purely functional approach and we're not gonna do any in place iteration whatsoever. If I wanted to – oh, get a clean board.

Here's a better function that illustrates how compact and dense. In many ways, it's a bad thing, but it's just a feature of the language, how dense Scheme code can be. I have two minutes to write this. I can certain do it.

What I want to do is I want to write a function that knows how to add up all the integers that are supposed to be in a list, okay. So I'm gonna assume that it's a number list. And so if I give you – let's just say sum of – and that's not a minus sign. It's actually a hyphen, so it's one token. And I give you this right here. You know it's supposed to be ten, I think, yeah, ten.

And the way that the Scheme functionality is gonna realize this is it's gonna say, "Oh, I have a non-empty list. That means I'm gonna add the CAR to whatever I get by recursively applying the same function to the CDUR." So the ten isn't so much a ten as it is a one plus a nine. The nine isn't so much a nine as it is a two plus a seven. Do you understand what I mean when I say that?

Okay, here's the implementation of this function. Define sum of – oops, sorry I did it, sum of – and I'm just gonna call it – I don't want to call it list. I don't want to get in the habit of naming my variables the same as built-in functions. So I'll call it num list just like that, okay. Does that make sense?

And what I'm gonna do is I'm gonna employ a couple of built-in tests. If it's the case that null question mark num list, then return zero. The if is exactly what you'd expect. It needs three parts to follow it, a test, an if portion, and an else portion. The else portion is technically optional, but I don't want you to pretend – I want you to pretend it's not optional.

Null actually comes back with true if and only if this thing evaluates the empty list. If it is empty, then trivially it's the case of all the numbers in this empty list add up to zero. Otherwise, what I want to do is I want to equate the original sum of call with the value that you get by levying plus against the CAR of the num list and the call to sum of the CDUR of the num list, okay. The headache really is just matching all the parenthesis, okay. But conceptually, this is the recur one that's in place to get this done.

Now you don't have to implement this recursively, but we are at the moment, okay. And we're always gonna opt for recursion over iteration in the Scheme segment of the course just to emphasize the functional aspects of the language. Do you understand how this is working?

It is just basic recursion, which is with the new syntax, okay. Synthesize the recursive result, get the nine back, add the one to it, and that's what this overall thing needs to evaluate to. Does that make sense? Okay.

So you have that. As long as I feed it one, two, three, four, it doesn't have a problem. If I feed it one, two, three, and then four is a list, it'll actually succeed in making three recursive calls, but only when it tries to levy a plus of the four is an empty list against a zero that it'll actually have problems, okay.

So it just does on an as needed basis the type of type analysis that is needed to confirm that the addition can be done between the CAR of the list and whatever was returned recursively, okay. Make sense? Okay.

I want to write a lot more of these come – today’s Wednesday, yeah, come Friday. I’ll write a lot more of these things. And then I’ll start talking about language constructs that are equivalent to the types of things we’ve seen in our C++ work and also in Assignment 3 and 4. Okay, have a good night.

[End of Audio]

Duration: 52 minutes

ProgrammingParadigms-Lecture20

Instructor (Jerry Cain):Oh, we're on. Oh, hey. Hey, everyone. I don't have any handouts for today. I still have enough material from Wednesday's handouts to keep you busy for the next year if you want. But certainly, the other lecture works. When I left you last time, I had just introduced primarily – in that, I'd talked about char and cuda and cons and all the little different – different atomic operations I want to deal with. Well, I'll give you an example of our first char cuda recursion problem. I had implemented this: sum, I'm gonna call it sum list, this time. And I will do – just call it num list. Okay, and I want to – even though there is actually exposed iteration in Scheme, I want to just pretend that it's not available to us. I want to take this purely functional and recursive approach to everything we implement, for the most part anyway. And so, if I want to be able to type in, at the prompt, sum of – or sum list, rather, something like this right here, I would like it to come back with a 15. Okay, if I want to go ahead and do something like this, I want it to come back with a zero. And so, what I'm going to exploit here is – I think it's something that's pretty conceptually obvious to everyone – is that the sum of this entire list is gonna be 1 plus whatever the sum of this list is. I'm gonna exploit the fact that cuda is available to me, and it's very, very fast to just kind of recurse on the tail of the list. For the implementation, is this right here. If it is the case that null of num list passes, or comes back with a true, then it evaluates this expression. Then this if evaluates to zero and the overall sum list expression evaluates to zero. Otherwise, I'd like to return the sum of the char of the num list, whatever that turns out to be, okay, and add it to the result of sum listing, the cuda of the num list. Okay, there's that; that ends the plus; that ends the if and that ends the define. Okay? Does that make sense to people? Okay, there are a couple of points to make about this that are not obvious. These two things, I think, if you trust the implementation and you just get really, really good at leap of faith and recursive programming, when you're programming in Scheme and Lisp, these are probably gonna work – I mean I'm sorry, you can look at these and just expect that this and that are the intended answers. And if you trace through the recursion, and use leap of faith, you will be able to confirm that. What I want to point out, here, is that what's interesting about Scheme is that, if you were to do this, at the command prompt you were to type in a request to sum list and you did this – let's say that I did hello 1 2 3 4 5, just like that, right there. It is going to fail; it should fail pretty much in any language, unless plus is overloaded to mean something when it's sitting between a string and an integer. But it is interesting to know how much progress it's going to make before it actually errors out and says, "No, I can't do this for you."

It is very literal in its following of the recursive recipe. Because it fails this node test right here, it doesn't return to zero. It calls char of num list and it prepares this hello to eventually be levied against this with a plus sign, but before it can actually evaluate the plus expression right here, it has to fully commit to the recursive evaluation of this thing right here. Okay? So what's going to do all of this work? And I'm kind of spinning – I'm putting a bad PR spin on it, but I don't want you to think about it that way. It's actually gonna go and assemble this 15, recursively. It's gonna bring the lift up the 15 as the evaluation of the recursive call, and it's gonna, at that moment, do a plus of hello against the 15 and only then, after it's assembled the 15, will it say, "Oh well, I can't do that."

and it'll issue an error. Okay? Does that make sense to people? The reason I'm highlighting that is because it underscores the fact that there is absolutely zero compiled time elements with regard to type checking. The type of compiling that goes on here – when you talk about compiling, it's almost always a fancier word for translation, and that makes sense in the context of C and C + +. Anything that's considered to be compilation in Scheme or Lisp is nothing beyond just parsing and making sure that the parentheses match, and the double quotes are matched, and things like that. Does that make sense? After it's been digested, and a link list has been built in memory to store this thing, it evaluates it and you're basically executing the program. Okay? So it's at runtime that tightness matches or, actually, detected and while the code is running, does it say, "Well, that won't work out." You may think that that's awful, but think about the equivalent in C of a heterogeneous list. It would have to involve void stars, okay, if you're gonna really support that kind of heterogeneity. And at runtime, it would also error out, but it would error out in a much more mysterious way, with those segfaults or those bus errors. Okay? At least this, right here, would tell you that there's a tightness match between the string hello and the 15. It might not actually have its real values, but it might say plus doesn't work when sitting in between a string and an integer. Okay? Does that make sense to you people?

Because Scheme is a runtime language, it actually preserves type information about all of the data. For the lifetime of this hello, the hello sits in memory and it's part of a small data structure that is tagged with something of an enumerated type saying, "This thing is a string." You have a little bit of a hint of that from a section problem that went out in week two. The first real discussion session, where I had people CACat note all the strings that were in a Scheme list, okay, the nodes in those lists are the things that really kind of back these types of data structures. And that they really are tagged with list, or integer, or double, or float, or Boolean, or a string and things like that. Okay? Does that make sense to people? Okay. So there's that. Let me just get a little bit more brute force in my recursion here. It doesn't – you don't always have to deal with lists. I will deal with lists, in a bit, in a couple of minutes again, but I could just define Fibonacci on n, okay, to be this, just to introduce a couple of other predicates. I could ask whether or not n comes in as zero. There's a built-in called zero question mark. The question mark is legal part of the syntax, just like it is part of null. It's supposed to stare at you and say, "There's a question being asked about the data that's in argument." And if that's the case, you just go ahead and return zero. Okay? Otherwise, you can do this. In which case, you can return 1. Okay? And otherwise, you can commit to the sum of whatever you get by calling fib against n minus 1 and fib of n minus 2. That ends the plus; that ends the f; that ends the first a f; that ends the define. Okay? It's kind of comical with all of the parentheses but it's much easier to do it in an editor, where you actually have parentheses balancing. Okay? I'm gonna rewrite this because I actually started writing this thinking factorial and forgot that I had two base cases. So I just kind of gracefully went through it like this is what I intended. But I actually would not – I probably wouldn't cascade my base cases like that. Okay? You can actually assemble base cases using or. I showed you how to do it with and before, so let me rewrite this: Define fib of n to be this. I would probably do something like this. If it's the case that – let's say or – you could do zero question mark n, or you could do something like that. I just want some symmetry in the base cases, like that. Then, I would just return n. And actually, I'm – yeah, that's right, n zero and 1. This, right there, is what gets evaluated as an expression if this, right here, passes.

Otherwise, I'd want a return fib of n minus 1 – I'm sorry, this is not minus. These are – no that's right. N minus 1 and fib of n 1 is 2. That ends the plus; that ends the if; and that ends the define. Okay, I'm assuming you're well behaved and you're not entering in a negative – not a negative number. There are ways to actually flag those, as well. There are trees and error function that you can invoke instead of plus or if or null question mark, you can call the error function and it prints out whatever string happens to be the argument, and it terminates the function call. Okay? Does this make sense to people? Okay. It's just weird getting used to this whole hyper parenthetical environment. And it's very difficult to remember that the parenthesis comes before the function call name, as opposed to afterwards. Okay? Make sense? Question right there?

Student:[Inaudible] after the n, do you need a parenthesis or, no?

Instructor (Jerry Cain):Which n is this?

Student: It's the [inaudible].

Instructor (Jerry Cain):Oh, this right here? No, this is actually occupying – let's say that the if, if not really a function, it's what's called a special form because it doesn't evaluate all of its arguments. But it takes either two or three expressions, afterwards. The first one is always supposed to be something that can be interpreted as a Boolean. So either it has to evaluate as a false or non-false. This, right here, has to be some expression that just evaluates if this test passes. It's actually completely ignored if this test fails. Okay? Does that make sense? And then, this is evaluated.

Now, you're thinking you need, maybe, something like that. Yeah, well that, actually, would try to invoke the n function with zero arguments. Okay? So n is supposed to be evaluated as a stand-alone variable. So now that I've talked about this if thing, I will underscore another feature. Just to show you how true this whole runtime idea really is in the Scheme language, if I do this – let's say that I just type this in at the prompt. I type in if it's the case that zero of zero – it looks pretty stupid, but I specifically want this test to pass. Then, I print out 4. Otherwise, I print out hello and 4.5 and the list 8 2, all added together.

Now, from a type standpoint, that else expression, that's gonna be evaluated if the test fails, it's completely nonsense. And if you know that if you type that in stand-alone, it would completely break down. Everyone believes me there? Okay? Do you understand that it's easy to parse it. It doesn't actually – when it parses this thing and builds a data structure behind the scenes, it does very little other than to say, "That's a token; that's a token; that's a token; and that's made up of tokens." and builds a list behind the scenes.

But only if it really has to commit to evaluating it, does it go ahead and see whether or not the code that's attached to the plus symbol actually can accommodate strings and integers and integer lists all as arguments, combined. But because the evaluation of this thing right here sees a test that passes, it goes and it evaluates 4. This is gonna print out 4, and it's not like it's gonna even bother analyzing whether or not the else expression would've worked out or not. Does that make sense? Okay.

That is emblematic of the type of thing. Some people say it's a feature. Some people say it's actually that's it's the type of thing that they don't like about runtime languages because a compiler would actually say, "You know what, please look at that because I don't think it could ever work out." Even if it's only called once in a million scenarios, I want to know upfront that it's gonna work out. A scripting language or a runtime language, like Scheme, or Python, which we'll talk about later, or Java Script, which we might talk about a little bit, about the last day of class, wouldn't analyze that at all.

So it's technically possible that code that you write doesn't get exercise for weeks, okay, because it just doesn't happen to come up in the workflow of a typical runtime scenario. Okay? Does that make sense to people? But it is emblematic and representative of the type of things that have very little or no compile time element to them. We're so used to very strongly compile time – I'm sorry, we're very used to strongly compile time oriented languages, like C and C + + and even Java, to a large extent, that we look at something like that and say, "Well that's just a blocker." And it really, technically, isn't. You can write whatever you want to here, if you know that it's not gonna be executed. And you wouldn't do that; I'm just illustrating a feature of the language. Okay? Does that make sense to people? Okay, very good. So let me do a couple more things. I want to get a little bit more intense with the type of recursion example I do because I want to illustrate how very conceptually sophisticated algorithms can be captured in a very small snippet of Scheme code. Again, it's like feature and a curse, at the same time. It means that the language itself is very articulate and terse in its ability to state and express algorithms cleanly. You don't need pages and pages to make a point; you need, like, four lines. Okay? Normally, you'd say that that person's a very good speaker but if actually a language, you say that it's clean or terse or expressive. What I want to do – and I love this function, so I'm happy rewriting it. I want to write this function called flatten. Okay? That's actually not an "h," flahen; it is flatten. Okay. And I want it to – I'll just illustrate how it works. I type this in at the prompt. If I give it this, it has the easiest job in the world. All it has to do is, it has to return the same list because the list is already flat. Okay? There are no nested lists, at all. But if I give it something like this, I want it to come back with this right here. Okay?

So I want it to conceptually, even – it will synthesize a new list. I want it to, more or less, look like it's taking the original list and taking all the intervening parentheses, and just removing them. And I'm implying that it has to either be integers or lists of integers; doesn't have to be. This could be the string 3; this could be the string 4. And I would just require that these things be strings in the output. Okay? It's almost as if you're doing this: something of a traversal through the tree, that's more or less implied by that nest of list structure. And you're just doing this inward traversal and you're threading through all of the integers like it's popcorn on a Christmas tree or something. Okay? And then doing this and saying, "This is where all the atoms live." Okay? And preserve the order in which you actually read them from left to right. Okay? Does that sit well with everybody? Okay. So you look at this and it turns out this would be very difficult, I think, to do in C or C + +, not because of the algorithm. The algorithm is actually kind of tricky, but in C and C + +, you would have to deal with the manual link list mechanics. You'd have to actually take lists and take the atoms that are in the nodes and, somehow, build new nodes around them and thread everything together. And it would be 50 percent memory management, 50 percent algorithm. Okay? All of the link list mechanics are, more or less, managed for you in Scheme, so there's 50 percent less to think about. Okay? Let me just implement this. I get to illustrate a new data structure. I also will say that I will not – I'm gonna avoid this. There's this one little nuance of the algorithm that kind of makes it more complicated. I just want to pretend that it's not a problem. If I give you something like this, in terms of the empty list, can either be considered a list or it can be considered an atom. In some dialects of Scheme, the empty list actually stands for null and false.

So I just want to pretend that this – that empty lists never actually appear in any of our examples. Okay? I don't want to worry about the edge case of having to deal with those. I just want to have this nice simple thing, where I'm always dealing with real atoms or lists that have some material in them. Okay? So I want to define this flatten thing, and I'm just gonna call it – I'll call it a sequence. Now I have three different things I want to think about: I am always – let me put a couple of examples up on the board. These are the three cases I want to deal with. One's a base case and two have recursive insights that we need. I'm either gonna hand you the overall empty list. That's different. Okay? Eventually, we're gonna account for everything and the recursion's gonna lead to an empty list. I'm just not allowing empty list to reside as elements inside the top-level list, okay, or anywhere in some nested list. Then I have this scenario; and I'm just gonna do this. I normally would commit to any extra values. Okay? I'm also gonna have this scenario. There's nothing in theory that prevents us from handling the scenario where the char of the list is, itself, a list. Okay? How do I flatten that? I do absolutely nothing; I just return the empty list because it's already flattened. It's basically like the vacuous list that, not only is it flattened, it's totally deflated. Okay? This right here, what I want to do is, I want to just a prepend, or cons the char because it's atomic and not a list, I want to cons the front – I'm sorry, I want to cons this one onto whatever I get by recursively flattening the cuda. Make sense to people? Okay. This is different. What has to happen is that I don't want to cons this element on to the front of whatever I get by flattening this recursively because I don't have a nested list right at the front. What I really want to do is I want to append the flattening of this to this right here. Make sense? Now there was – that didn't technically it all because it might be this. You know, it could be this ridiculously hyperlinked structure that happens to be sitting at the char. So what I really want to is, I either want to cons this really simple atomic element to the front of the flattening, or I want to append the flattening of this to the flattening of the char. Does that make sense?

Okay. So you have to be clear on the difference between cons and append. I'm not gonna do cascaded ifs, like I did right here. I'm gonna use another data structure. There is something called cond. I like this. There's no real analog – there's a little bit of an analog to this in C and C + +. It's kind of like a switch statement, but it's a switch statement on Boolean tests, as opposed to scalar values. What's presented to cond is a series of pairs. The first item in the pair is a Boolean test; the second item in the pair is the express that gets evaluated if the test actually passes. So what I would do up front is, I would ask whether or not the sequence is null. And it that is true, then the entire cond should evaluate to this thing right here. So that's that; that ends that; that's the pair. I have that right there. So this opens up the list of pairs; this opens up the first pair; this happens to be associated with the test that's inside the pair. Okay? Does that make sense to people? Okay. It's just a lot of parentheses; it's more – it's easy to get, just once you to type it in. But the first thing I want to check for is whether or not the sequence is null. Okay? Make sense? I'm actually gonna handle this scenario as a second clause right here because it happens to be a built-in predicate that tests whether or not something is a list. If I fall past the null sequence because it's not null, then I know that I have at least one element in there.

So what I want to do is, if it's the case that the list predicate – that's another built-in; it was in the first of the two handouts I gave in on Friday. It's just is this predicate that says, "Is the thing that's serving as my only argument here, is it a list? Because if so, I want to react differently. Is the sequence actually," – I'm sorry, "Is char of sequence – char of sequence, is that a list?" If so, what I want to do is, I want to flatten the char; I want to flatten the cuda and I want to append them. Does that make sense to people? So what I would do is, I would call the append function on the flattening of the char of the sequence, and the flattening of the cuda of the sequence. That ends the flatten; that ends the append; that ends the entire order list. Okay, the entire [inaudible]. There's enough parentheses, had to be smushed in here between u and the end of the all editors, to balance that parend, right there. Okay? The third scenario is supposed to be the else scenario. Okay? I've seen a lot of things happen here. Normally, you want to make sure that the last in the sequence of tests that gets evaluated is guaranteed to evaluate to true. Okay? I've seen some people do this because that certainly evaluates to true. That's the Boolean constant true in Scheme. There actually is a key word that only makes sense in the context of the con statement. You can't use it outside of a con statement, which is actually interesting from a syntactic standpoint. You can't always – you don't see that in a lot of other languages.

You can use the keyword else right there, which basically, is synonymous with true in this context. What you would do is, you would cons the char of the sequence, which requires no flattening because it's not a list, to whatever you get by flattening the cuda of the sequence. [Inaudible] Okay. You guys get what's going on here? Yes, no? Okay. This right here, cond, it actually does expand; it is this macro that syntactically expands to cascaded ifs. Okay? It evaluates this test; if it passes, then the overall cond evaluates to that. Otherwise, it falls through, and evaluates this test, this test right there. If it passes, then it evaluates this thing; otherwise, it ignores it, and falls down here. Else always evaluates to true, so it would certainly evaluate this if it got this far. Okay? If, for whatever reason, you put a test there that also failed and it's potentially the case that all cond tests fail, then the – not the return value, but the cond – what the cond expression evaluates to is not actually defined. You can't rely on it..

Most implementations just return the empty list, okay, but I don't want you to code that way. I want you to make sure that any con statement that you ever evince to implement an algorithm, has this default case. This is the equivalent of default in a C or C + + switch statement. Okay? I just want you to know that one of your expressions is gonna work out. Okay? Make sense? Now, you notice that there's really no functionality – I mean, think about all the functions that are used: There's cond, of course; there's null; there's list; there's append; there's char; there's cuda; there's flatten itself. At no point, do I invoke any functionality that's specific to ints vs. Booleans vs. strings, which is why it's perfectly happy to flatten lists that have a variety of atomic types inside. Okay? Does that make sense to everybody? Okay. Question over here? No. Okay. What else is going on? Okay. You guys are good here? Okay. Questions on the left. What's up? Yup.

Student: Wasn't that thing you write an [inaudible].

Instructor (Jerry Cain):After else? This right here is cons. Is it just the word you're curious about, or what does this do? I can write it out more cleanly if it's not clear.

Student:[Inaudible]

Instructor (Jerry Cain):This is cons. Remember what cons – I talked about – what's that?

Student: What does it do?

Instructor (Jerry Cain):Cons, basically – cons is short for construct and it takes an atom; and it takes a list; and it evaluates to a list where this is the first element and everything in the list comes afterwards. Okay? So it's, basically, this built-in prepend function. And if you think about it from an implementation standpoint – we're gonna talk about the memory model, next week, as to how this thing works. But if you have a handle on the front of the list, it's very easy to change what the front element is. Which is why char and cuda and cons, which manipulate, either extract or update the front element. This is me, holding a link list here. Why those things are supported operations is because they can run in constant time. Okay? Yup?

Student: So at least in our implementation of Scheme [inaudible], arguments can be evaluated before they're passed into a function?

Instructor (Jerry Cain):They actually do, yes. That is common with, I think, most languages. Not – actually, I don't know of any specific examples, specific languages, that defer the evaluation of arguments until after the char. Like, there's some versions of, well some older dialects of Java script. And I think certain features of Pascal, where they had some kind of copy – like copy – passing like copy semantics. They had some, like, kind of parameter passing Scheme that I'm not familiar with because if was really before my time. I just know it exists; I just don't know it very well. But in our world, in our Scheme implementation, it is the case that the arguments are evaluated before the function is invoked. Okay?

Everyone doing good? Okay. I have – I want to motivate another example. You have tons of examples in the two handouts I gave you out on Wednesday. I'm not gonna go over all of them because I say that, for every concept, there's probably three examples, and I just don't want to cover all three, when I think one suffices for lecture material. I do want to implement one other function that's clever in its use of or and and, and recursion to determine whether or not an integer list is actually sorted in a way that it respects the less than sign. Okay?

So let me write this function. I want to write this function called sorted, with a question mark, just because I can do that. And I'll just be consistent with the way some of the built-ins use it. And I will allow it to take this thing that I'll call a num list. I probably should call it an integer list because I'm really thinking about integers, specifically. But what I want this to do, is I want it to return true if the list that's fed as an argument is already sorted in non-decreasing order. Okay?

So something like this: Sorted – sorted of something like 1 2 2 4 7, should return true or something equivalent to true. Okay? And if I try to walk the system and do something like this, certainly it's close to sorted in some informal – by some informal definition of close. But because it dips once, it's technically – excuse me, not sorted the way I want it to be. Okay? So there's this; this will come back with a false. I want to implement this with the understanding that the num list really is numbers. So that I can compare things with the less than sign.

Turn it and you can compare strings for equality and inequality. You have to use different flavors of less than and greater than, and equals to. You have to basically use the functions that are designed, better designed to work on strings. So there's that. I've erased this because I want to make room for it. The base case is actually – I don't want to say it's too – it's very clever. It's actually kind of, it's kind of obvious.

But all lists of length zero, and of length 1, are already sorted. Okay? So what I want to do is: I want a return. It doesn't – it almost looks like there's no – I love this implementation because it looks like there's almost no base case. But I exploit or and it's short-circuit evaluation. Okay? And so I really do have a base case here, even though it's not expressed in this pure, if base case, do this scenario. I want it to return the truth or falsity of the following: That either the length of num list, this is another built-in – and I know I'm spraying built-ins at you – but they're all kind of obvious and you should know that they just exist in any language that has lists as a built-in.

If the length of the num list is less than 2, then you have your answer, and you could care less what the second of the disjunct is gonna evaluate to. Okay? If this passes, you're done; the thing is sorted. Otherwise, you actually know that, not only is there a char, but there is a char of a cuda as well. Does that make sense? Okay? So if I get this far, and I'm evaluating the expression that happens to have an open parend that I just drew right there, then I know that I have at least two elements.

So what I want to do is I want to confirm two things. If I have these two elements, I have x and y. They could actually really be x and y, I mean they just have to valuate to something. And then, I have all of these other things. I need to confirm two things to decide if this thing is actually sorted. I have to see that this thing right there – that x really is less than or equal to y. And that the cuda passes the sorter predicate. Make sense? Okay.

So there's that. So I have this and right here and I want to do this. Less than or equal to of char of num list and – I want to do this; I want to – I'll this – I'll invent – I'm not inventing; I'm actually using another function that's built-in. It's kind of funny that it exists. You have no idea what c a d r is, but you kind of have an idea as to what it's probably getting at. This right here is obviously supposed to be the first element. When you see something like c a d r, you're supposed to actually pretend that this is really two words nested, like cascaded; that it's supposed to be the char of the cuda. Does that make sense?

If you wanted to put char of the cuda with extra nesting calls, you could do it. But if, for whatever reason, you want to get the cuda, of the cuda, of the cuda, of the cuda, of the cuda, that happens to be a built-in. Okay? If you know, structurally, that you want the char of the cuda, of the char of the cuda, and because of the way things are nested together, you want to do that, then you can use that instead. Or you can go forth with the actual exposed char in cuda calls; that's all this evaluates to anyway. Now, you may wonder whether or not, you know, something like this just parses and it figures it out. It does not. It actually stops at four, at least as far as I know. That's the built-in; that's where the built-in stops.

Okay? If you really do programming in Scheme for a living, like you're figuring out when c d d d a r is relevant, it's almost as easy as figuring out when char is relevant. So it's not like you have to go through and, like, understand the full 2 to the 4th different permutations of this right here. Okay? You just want whatever; you just want to know that they exist so that, when I write something like this, you know that I'm really trying to get to the second element. Okay?

That balances that; that balances that call. I also want sorted to just work out on the plain old cuda, just 1 d, of the num list. Ends the and; ends the or; ends the define. Okay? There are some implementations of Scheme, not ours, that understand that matching all the parentheses at the end is a little bit difficult. Some of them have overloaded. Ours does not, but this is funny. Some of them have overloaded to say, "Oh, whatever, just use the square bracket." and it'll just round them out to figure out what it would have been had you actually had the patience to count your parentheses. But unfortunately, ours does not do that. So you really do have to commit to this thing called counting, okay, to actually match everything. Okay? Does that make sense? This is nice because if this thing fails, it doesn't bother with the recursive call. So there are two short-circuit evaluations here that really do save us time. It is prepared to march through enough recursive calls to come back with a true and it has to do that. It has to be exhaustive in its search of the entire array to come back with a true because it doesn't want to make any mistakes. But it only has to find one flaw in the array, for it to return early with a false. Okay? Does that make sense to people?

Now, it turns out that – I didn't know this until I put it in the handout – but this particular flavor of Scheme allows less than and less than or equal to, to actually take multiple arguments. We're used to seeing something like this, and when we look at that, as soon as you get used to the prefix notation for function evaluation, you look at that and say, "Okay, that's really asking a question as to whether 1 is less than 2," and it comes back with a true. It turns out you can do this as well. I'm not saying you should exploit this, but the is sorted for less than or equal to, for instance this right here, where I have two sixes at the end. This also would evaluate to true. I don't want you to have to remember this; this isn't really intellectually engaging. But it's kind of neat that it was smart enough, or at least robust enough, to recognize that less than or equal to doesn't have to be this implicit binary function. It could really just be a request to see whether or not all neighboring pairs in a list actually respect the common notion of what less than or equal to means, rather. Okay? It doesn't – not surprisingly, it doesn't work for equals or not-equals. Okay? I mean, I guess it could've worked but it's, like, why would you expect them to put down a list of length 20 and ask whether or not they're all equal to one another. Okay? But it does work for all of the inequality operations. Okay? Make sense? Okay, that's fun. So what I want to do here is, I want to speak a little bit about – I'll be more sophisticated in my explanation of this next week, when – but I'll probably draw the same pictures next week. I just want to get to them now so that I can really talk about how function pointers work, okay, the equivalent of function pointers in Scheme. Do you understand that, algorithmically, this is pretty much spot-on as to how it would confirm whether or not any sequence is sorted, except that it's constraining it to be number specific because of its use of that right there? Even this is fine. That's there to compare lengths of a list of two and it's gonna be there regardless of whether we're dealing with string lists, or complex number lists, or whatever. Okay? But this is the thing that's actually requiring that all of the elements in the list actually be numbers. Okay? Let me just give you a little bit of an idea as to what happens when you invoke a function. I've already spoken informally about it, but let me actually draw some pictures.

When you invoke char of 1 2 3 4, a couple of things happen. It actually digests this token, okay, and it happens to occupy something of the leading slot in the series of slots that make up a link list. Does that make sense? And you know, by looking at it, that that is a symbol . It's supposed to be a test of functionality that tells you how to digest and manipulate the remaining arguments. Okay? It's very close to this; this thing itself points to a list. And it has some tight tagging so that it knows whether something's a list or an integer, or whatnot. But this points to a 1, points to a 2, points to a 3, points to a 4, etc. Okay? Technically, that thing's quoted; so there's really a quote function there, as well. But that's kind of the over-arching idea of what the memory model would look like, for this particular call. Does that make sense? Now, without being too sophisticated about it, I think it's reasonable for you to understand. I just drew a symbol table of functions there.

And among the functions that are defined here, there actually is an entry where the key is the string char and it's associated – and I'll just write it this way – with code. Okay? That code that should be invoked whenever char is consulted as the function that should be guiding evaluation. Okay? So what happens, and it really does do this behind the scenes, it digest this; it builds this as a data structure. It does any recursive evaluation of arguments that it needs to do, but it will be suppressed here because of the quote. It then goes and finds the funct – the code that's associated with this and then, there's some general meta-code thing, behind the scenes, that figures out how to manipulate the arguments, and produce a result based on the contents of this thing, right here. Now, you've seen how we define, not char or cuda but how we define flatten or sorted question mark or sum list and things like that. We exactly associate the definition of something like flatten or sorted with code that's expressed in link list form. Does that make sense to people? Think about it; you use lots of parentheses after the define of num list. So this, right here, actually is stored in memory much like these things are. Okay? It's just understood to be an in-memory representation of the recipe that needs to be followed, and a series of instructions as to how to manipulate the remaining arguments of the function invocation. It really is this in-memory representation of the functionality. Okay?

So it's almost like – I'm trying to think of the best analogy of this. It's almost like you're reading, from a file, the series of instructions that are allowed. Okay? And you store those instructions in memory and associate them with this keyword, right here. And that there's some way, behind the scenes, that it actually uses this to guide execution of this type of execution statement, right there. Okay? Does that make sense? Okay. Now, there's much more detail to this; I make it sound like you could just write it in an hour. It's not the case at all; it's actually pretty, pretty challenging. But nonetheless, that's the over-arching idea. The reason I'm saying that is because this thing right there, it self-evaluates to this. It's because it occupies the 0th slot, as opposed to the 1th, or the 2nd or the 3rd slot, okay, that this right here is assumed to be a variable that's associated with code as opposed to just raw data. Okay? So in the same way that this evaluates to itself because of the quote, this evaluates to the code. Okay? There is actually a way to type in an anonymous function, right there, and have it guide execution. I'll show you that on Monday. Okay? But char really, here, it does evaluate to a block of code; it's like this. In C or C + +, this is, at compile time, taken to be an instruction as to where to jump to in memory. In Scheme, it's taken as an instruction as to where – what symbol to look up in the global symbol table of definitions, and assume that there's code associated with it. Okay? The reason I'm saying that, is because at the moment, we've implemented this thing called sorted so that it hard-codes this in right there. Okay? If I want to generalize this, okay, and go all, like, vector sort on you or, like some kind of – be as a polymorphic in my support of a generic sort is possible, you do it in Scheme. You do it with something that's equivalent to a function pointer. Okay? But rather than passing in the address of the function, you pass in the function itself. Okay? When I say that, you actually pass in the code as a parameter. Okay?

Let me show you what that would look like. At the moment, sorted question mark of the list 1 2 3 4, just works according to that recipe, and comes back with a true. What I want to do is this: I want to be able to invoke sorted, so that it can take 1 and 3 and 5 and 7.

But recognizing that this could've been an array of – I'm sorry, a list of strings or a list of lists, I actually am gonna pass in that. It's a stand-alone token; there's no spaces in between, so it knows that less than or equal to is the thing that's being passed in is the 2nd argument now. You haven't seen the new prototype for sorted yet, but you can imagine that there's gonna be some variable that catches whatever this evaluates to. And it's gonna evaluate to the code that it's associated with that knows how to compare two or more elements, actually technically one or more elements, okay, to figure out whether or not all of the elements respect, in the way they're listed, respect the less than or equal to predicate. Okay? If I want to do this – this would, just presumably, come back with true because we're assuming that less than or equal to does the right thing – sorted a b d c, and I pass in this thing called string less than, that just happens to be function that's the equivalent of less than on strings. So it operates like normal [inaudible] less than in C + + strings. I'm sorry, less than – yeah, that's right. Okay? I would expect this to return false for the right reasons because this, right here, is failing it. Does that make sense? Okay. So I would expect this to come back with a false. So it turns out that the implementation of this sorted thing doesn't have to change much at all. I am gonna rewrite it fresh, even though it's in the handouts because I don't like changing code unless it's absolutely ridiculous not to, to just change it. But I'm smushed on room here, so let me just write it fresh. I want to define sorted and I'm gonna take – I'll just gonna call it s e q, so I'll have a shorter word for it, and I'll just call it comp. Now, the way I'm invoking those Schemish expressions over there, I'm expecting that something has been evaluated, and it's evaluated to code and it's being pushed on to the 2nd of these two variables right here. Okay?

So locally, this c o m p, it's almost like it's the name of a function that exists elsewhere. Okay? So I'm gonna implement it to be core – I don't know where that is – or less than length of sequence of 2 and – what's with the c? Sorry, char and cuda. And I want to put c o m p right here. Char of sequence, char of the cuda of sequence, and then, at the same time, I need sorted on the cuda, and finally use a c of sequence with the same comparator, an and, and the or, and the define. So if, at a first pass, the kindergarten explanation here is that comp is function pointer, okay, that can assume the place of a function name in the implementation, that would probably be enough to just make you understand what's going on. Okay? Technically, it is a variable on – that actually has attached to it, whatever this, or this evaluates to. And it doesn't have to be a built-in; it can be anything that we define that knows how to compare 2 or more elements or just 2 elements, in this case, exactly 2 elements in this case, and come back with a true or false. Okay? Does that make sense? So this, right here, it's attached to a list, like 1 3 5 7, or a b d c. This, right here, is also attached to a list. That because of the way it's invoked and it appears as the 0th element in a function call, or in a list, that it's supposed to be associated with code, or a link list that knows how to guide execution and manipulation of everything that follows it. Make sense? Okay, that's great.

So there you have that. There are a couple of other examples in the handout that deal with this. I'm not done with the function pointer idea. I have so many analogs to things you're just accustomed to, in C and C + +, the notion of mapping, the notion of anonymous functions, and client data, and all that kind of stuff. There are all these cool things in

Scheme you just have not seen in other languages. Okay? And they exist in extensions to the languages, like extensions to C and C + +, but they're not part of the core language. There's something that I'm gonna talk about, on Monday, is core to Scheme and it doesn't really exist in the core of any of the languages that you've seen before. Okay? So I'll have that; I'll have more examples with function objects, which is what those things are really called. And then, I'll talk about mapping, filtering, things like that. Okay? Have a great weekend.

[End of Audio]

Duration: 52 minutes

ProgrammingParadigms-Lecture21

Instructor (Jerry Cain):Hey, everyone. We're online. I have a good number of handouts for you today; they're all going out. They've been posted since yesterday. Two Scheme handouts for material that I, just briefly, touched on in the last five minutes of last Friday, and today, and Wednesday, and Friday. Still, probably, we'll be covering Scheme. Your next assignment goes out today; it's not due for a while. I'm not making it due until the Wednesday evening after the weekend. History has shown that the hardest part about this entire assignment is getting the first of the eight or so functions that you have to write, written. Because that means you have to get used to CALA, which is a development environment, and getting used to all of the parentheses, and understanding how to test your code, and things like that. After you get that working, then the assignment is, actually I think, on the easier side, as far as 107 assignments go. But recognize that the first function, for a lot of people, is actually a lot of work. It's not unheard of for some people to spend, like, an hour or to 90 minutes on just the first function, which is like seven lines of code, and then, after that it's just smooth sailing. But they have to iron out the rough patches with the first function, then it gets easier. Okay? We will have a discussion section, tomorrow. The examples in this section handout for tomorrow are being passed out, right now. It's actually a pretty difficult set of problems. They're old final exam questions that I've given in past quarters. But nonetheless, I mean, it's good material for understanding all the little quirks and neat things about Scheme. So I'll let you work with those tomorrow, with Ben Newman, who will be teaching it. What I thought I'd do is, I thought I'd introduce you, a little bit more formally, to the development environment and show you how that's gonna contribute to your Assignment 7 experience. Okay? So let me bring this down, create some mood lighting, and let's see where we are with regard to getting everything to fit on the screen. Please, please, please, please, please let it fit on the screen. I don't know what that's about, right there, a little mushroom. It's a microphone, I think. Oh, there we go. Okay, so let's look at – make sure that this fits on the screen. I'm not sure what people in TV-land can see. It looks like they can see my entire computer screen, but it's projecting larger up there, for some reason. But as far as I can tell – we're working on it. This is you driving, right? It's actually not a disaster, if you just wanna leave it. Yeah, 138's good. That's me up top. Let me just – we'll figure it out. Okay. Everybody see that, in the room? See, all they see is Jerry at the bottom. So on the lanes, and on the pods, where I know I've tested it, it also should be working on this as well. If you want, you can deal with the Scheme environment like I introduced, last week, or you can type in 4, and have it confirm that it's 4. Or you can type in hello and have it confirm that it's really hello. But if you want to do things that are meaningful, from a mathematical standpoint, you can do that and come back with 21.

So this is shell-like environment, where you happen to be dealing with the Scheme language. Now, I don't want to imply that, somehow, programming and Scheme is all command line driven; it's not. Normally, what you will do – let's spell quick correctly – let's – sorry, exit. Let me do this – let me, actually – well, this is huge. Oops. There we go. Okay. That's a little too small for this screen, though. Hold on a second. It's gonna have to be this. Okay, good enough. What I want to do is, I want to split this; I want to bring this, and open up a shell, down below and I'll play with CALA right here, in the lower half. But what I'm gonna do up here is, I'm gonna prepare a little coat-snip at the one I finished with on Friday, where I would define, in place, this thing. If you recall, I called it sorted. And I said it was capable of sorting the sequence, but in the end, I said we're gonna deal with Scheme's equivalent of a function pointer, which we call a function object in Scheme. And the intent here, is that I'm defining this two-argument Scheme function that takes the list and then lock them into a data type. I want it to be heterogeneous by specification, but I don't care whether it's a string list, or a list list, or an int list, or what have you. And then, I pass in this thing that knows how to compare pairs of elements that reside in that list. And this is what I wrote last time. I wrote, "Either it's the case that this list is so small that it can't help but be sorted, or it's the case that this CNP, when levied against the char of the sequence and the cuda" – I'm sorry, "the char of – of the cuda of the sequence" – is that fitting on the screen? Yes, good. There's that, "and it's the case that, at the same time, sorted cuda of sequence, with the same comparison function, kinda just works out with this recursively." Okay?

So I can bring this down so you can see it. And that's what I left you with on Friday. Does that make sense? How's that turning out on the screen? I think that actually looks okay on the screen, so TV people, people in TV-land shouldn't be yelling at me. Okay? I'm gonna save this file, and there's a command in CALA, this actually common in most Scheme interpreters where you can load a code file. So in the past, we've prepared dot C and dot CC files ahead of time, and then gone into a shell and actually typed make to build the executable. What we do now is, we actually prepare our Scheme functions in a file that's suffixed by dot Scm for Scheme and then, we actually load them into the interpreter, using this load command that I'm doing down here. Does that make sense to people? Yes, no? Okay. So if this all works out, do that, and then I can type in, "Is it the case that this list is sorted according to that predicate right there?" And hopefully, it comes back with the truth. And it does, the green – the green T. Okay? If I do this, and I jump all over the place, and I ask whether or not it respects the greater than and equal to predicate, it should come back with a false, and it does. Okay? Make sense to people? Okay. So that's the over-arching idea as to how you interact with a Scheme developmental environment. I give you a dot SCM file for the "Where am I?" assignment, that's going out today. It has tons of code that's already defined in there, but you're gonna go through this – this update the Scheme file, save, and then type load in a parallel Kawa shell, okay, just to kinda bring in whatever code you've most recently written. Okay? Does that make sense to everybody? Okay, very good. So there's that. I will put this to bed; I may come back to it. That's fine. But I want to write code on the board because I think it's nicer to create the code, than to have it prepared ahead of time.

There is this one idiom from C + + with iterators, and from Assignment 3 with vector math, that I want to see the equivalent of in Scheme. Do you understand that, if I write this as a function, sum all – actually, I don't want to do that, I'm sorry. Let's say, not sum all. Let's say double all, and I give it this list. And without writing code, I think it's pretty clear that the output of this should just be 2 4 6 8. If I have another function, called increment all, and I type in this right here, I expect it to spit out 2 3 4 5. Okay? Do you understand how those are algorithmically similar? They both visit every single element in the list, using char cuda recursion. Okay? They both apply some functionality to each char, okay, but the output is a list of exactly the same length as the incoming list where each thing that ever served as a char of a cuda is transformed by either the double operation, or the increment operation. Does that make sense? Okay? Except for the fact that this is Scheme, it kind of screams vector Map. Does that make sense to everybody? Okay. Well, it's not like Map, as a verb, was coined for C and C + + purposes. It actually exists everywhere, and rather than doing this, what I could do is, I could define a double function, which is not a built-in, but I want to just do this and I can equate the double operation of an x with that as an expression. I could just do it on one line; I don't have to do it over multiple lines if I don't want to. Okay? I can also define the increment function to be associated with this successor thing. Okay? Does that make sense? There is an operation in pure Scheme called Map, and it takes 2 or more arguments. We're gonna deal with it as if it's a pure 2 argument function. The second argument – I'm sorry – the first argument to Map can just be the name of some previously defined function, whether it's a built-in or something you just defined. And then, you can specify what list you should be Mapping over. Okay? And the result of that call, right there, if I typed it in at the prompt, would be 2 4 6 8. If I do the same thing with the Map operation, but Map incr, this function over the list instead, then I should get out 2 3 4 5, and be done with it. Okay? Does that make sense? Now, Map is a little bit more sophisticated than I'm making it out to be here. I'm making it look like it Maps unary functions over single lists. Okay? That will change as we get a little bit more experience with it, but I want you to understand that this is kind of the more functional approach to this up there. If you actually implement that and that, then you're implementing it in terms of exposed char cuda recursion, with both implementations.

Does that make sense? Yes, no? Think about what the implementation of that would look like. What I could do, instead, is I could either use the built-in Map, or I'll just pretend for a minute that the Map function isn't a built-in and we're gonna define it in a second. We can extract the notion of a Map to use char cuda recursion regardless of what this function turns out to be. Okay? And recognize that this can be passed in and caught in a variable like CNP was in the sorted question mark predicate function. Okay? So I'm actually going to – I'll give you the full story on Map. If you want to, you can – you can Map a lot of interesting functions over lists, if you want to. If I want to Map the char function over a list, I could certainly do it. It better be a list of lists, and that would spit out the list 1 4 11. Okay? It's still the case that it transforms the list, right there, of length 3 into another list of length 3. And it uses this operation right there to figure out how to transform each thing that comes up as a char in the recursion to some element in the final list. Yes? Okay. If I want to do this: Map, cuda 1 2 4 8 2 11, and I can do that. This would give me the list 2, the list 8 2, and the empty list. And I, once again, get a list with 3 top-level elements inside. Okay? The Map function – we're not gonna implement it this way, but I just want to advertise what it is. It's kind of neat that it can do this. If I want to Map the cons function – now, cons is different than all of the other examples, or all of the other Mapping functions I've chosen in the past because it's not unary. Okay?

Well, cuda and char and ink – inker and double are all unary functions, which makes them compatible with a single list to follow it. But if I want to, with a real Map, I want to Map cons, what I can do is, I can provide two lists, where one list provides all the first arguments to the cons calls, and the second provides all the second arguments. So I could do this, and what happens is that the Mapping function takes cons and applies it to the 1, and the 4 list to synthesize the first element. Okay? The next element, the second element of the product, has a 2 cons onto the front of the empty list. So the product of this thing, right here, would be the list 1 4, the list of 2 just by itself, and the list 8 2 5. Okay? Does that make sense? If I wanted to, I could Map the plus function over as many lists as I want to and now, I have to add these together, but it would give me 1 10 and 16. The output is a list of length 2 because all of the input lists are of length 2 and this assimilates all of the chars of the 3 arguments into one sum and then it assimilates all the chars in the cudas into the second argument and then it realizes that all the lists are empty. Okay. Map is robust enough that, if you give it lists of different lengths, if I were to sneak in a third element in the middle list there, it wouldn't freak out. It actually just terminates when the smallest of all the lists reaches its end. Okay? So that particular one would be ignored because the other lists or at least one of the lists is of length 2. Okay? That make sense to people? Okay.

This is the purely functional way of actually visiting, or transforming an incoming list, normally, or a pair, or a triple of incoming lists into another list, okay, that's completely unrelated from a memory standpoint, but it kind of emulates what foreach does in – the foreach algorithm does in C + +, and what vector Map does from our Assignment 3. Okay? We can implement Map. I'm not gonna implement the binary version yet. I just want to use what we've learned, based on the last ten minutes of last Friday, and in the first few minutes of today's lecture, to implement our generic Map function. I can define Map, but I'm not gonna call it Map because that's the name of a built-in. I'm gonna call it my – and I'm gonna say unary Map to emphasize the fact that I'm implementing a subset of the real Map function. Okay? And I'm gonna pass in an actual function and I'm gonna pass in – I'll call it a sequence. Somebody asked about dot dot dot, last time and the equivalent of dot dot dot. Do you remember that question? Okay? I will use this to motivate the equivalent of dot dot dot, when I implement the generic version of my Map, later on. Okay? But right now, I'm just dealing with one unary function, and I'm dealing with one list, which I'm calling seq. Okay? If it's the case, regardless of what f n turns out to be, if seq is null, then I return the empty list, and f n has nothing to do with it. Otherwise, what I want to do is, I want to cons onto the front whatever I get by Applying f n to the char of the sequence, okay, to the result of calling my unary Map, mum, of f n against the cuda of the sequence. That ends the my unary map; that ends the cons; that ends the f; that ends the entire definition. Okay? Except for the word f n which, in this case, is the name of a local variable that's ostensibly bound to a function, this would, more or less, be the implementation of double all and increment all that I crossed out, up there. Okay? I would've hard-coded double and incr into two separate implementations. I'm trying to abstract a way and say I'm gonna make this general enough that I can pass in the function that I'd like to be applied to all the elements inside. Okay? Does that make sense to people? Yes, no? Okay. So there's that.

So this Map function – you may say, "Is it all that useful?" And the answer is I think it's kind of useful. I want to show you a different way of actually flattening all of the elements in the list. This is, I think, fascinating how it does this. Okay? If I put this list up here – we're revisiting the flatten problem that we dealt with last time. Do 1 2 – actually, you know what? Let me just – let me introduce a couple of other functions, before I do this. I want to get to this, but I'll get there in five minutes. There are a couple other functions that are built-in to Scheme, that I want you to understand. There are – one's called Apply; one's called Eval. Eval is easier to introduce, and it's a little bit quirkier. Do you understand that, when I type in open parend plus, space one, space two, space three, space four, space five, close parend, that the interpreter actually reads it, tokenizes it, recognizes that most of them are integers, and it somehow, figures out how to invoke the plus function, okay, based on what I typed in. Eval recognizes the fact that everything, at least from – whether it's sorting a file, or it's typed in at the shell, comes in as a stream of text that needs to be tokenized and evaluated. Okay? If you type in – this looks weird, but if you type this in, with that quote there, that suppresses evaluation. Right? This would actually come back, and it would list this as the output of that operation. Eval, even though this isn't – you wouldn't type this in this way because you would just go with the more direct way of adding the first three integers – but if you type this in like that, whatever its 1 argument is, it evaluates it. In this case, it evaluates just to the list constant plus 1 2 3 because it's quoted, and then it evaluates it as if it were typed in at the command prompt in the interpreter. Does that sit well with everybody? So this would, slightly less direct means, be a way of actually printing out the sum of the first 3 integers. Okay? Now, this is certainly the less common of the two operations. Apply is cool; Eval is neat because it advertises the fact that functions and data are really exactly the same thing in Scheme; that everything is expressed as a list. Okay? The function – the code that implements Eval in the interpreter for this Eval, is the very code that gets invoked every time you hit enter, okay, at the interpreter. It reads in the text, up to that – to the last matching parentheses, and then it evokes some Eval function, behind the scenes, to get it to produce a 6, or whatever it wants to produce. Okay? Apply is a little bit different. Apply actually allows you to specify which function should be, basically, pressed against all of the arguments that follow. That's another way of computing 6 in a less direct way. Okay? But it won't always be the case that we know that the last argument is the list constant 1 2 3. It could be something that's procedurally or functionally generated. Okay?

What Apply does is it always takes exactly 2 arguments. The first one is always supposed to identify some block of Scheme code. What follows it is always supposed to be a list of the type of data that can actually be levied against by this type of operation. So what Apply does is, it effectively takes the plus; it cons it onto the front of this thing and then, it evaluates it in this Eval sense. Okay? So this would come back with a 6. Okay? More meaningful version – more meaningful example of where you would use Apply? If I just assume that I don't ever have an empty list, if I want to define the – let's say the average function. Literally, I want to compute the average of all the numbers in a list, and I'll just say num list. And I'll assume that they're all doubles, so that I don't have any fractions and truncation and things like that to deal with. So I have all of these scores, like 39.5 and 40, and 29.5, and all these types of scores we saw on the mid-term. If I wanted to, I could write my sum all function, using char cuda recursion, okay, to synthesize the sum of all the scores, and then divide by the length of the list, or I could do this. That right there. Okay? That's a more meaningful, less contrived example because I don't know what num list is. And if I really do pass in num list as a list of scores, I have to, somehow, get the plus to be effectively onto the front of that anyway. I could use char recursive cursion. I can't use Map because Map transforms a list of length n to a list of length n; I don't want that. I want just to take a list of length n and produce a single number out of it. Okay? So the Apply function is this quick way of actually saying, "You know what, I really wanted a plus sign onto the front." or "I wanted the function to be the very first element of this list that's right here. So can you just, please, tuck that onto the front and then evaluate it like it was there all along?"

Okay? Does that make sense? That is a much cleaner way to do it. The drawback of exposed char cuda recursion is that it's asymmetric, and it advertises the asymmetry that's involved in visiting the char of the entire list first, and then the char of the cuda second, etc. Whereas, both Map and Apply make it look like all of the elements in a list are peers, and the simultaneously contribute to the overall effort – or the overall computation. When you Map the double function over the list 1 2 3 4 5, all the way through 10, it's like it just dumps out the list 2 4 6 8, etc., okay, in one fell swoop, without actually exposing the fact that there's char cuda recursion involved. The same thing with Apply. Rather than actually doing this sum all function, where you recursively, you know, compute the sum all of the cuda and then add, with a plus sign to the char of the entire list to that, you just say, "Okay, num list, all of the elements, you're participating in a plus." Okay? "I'm applying the plus to all of you. Cooperate and come up with your sum." Okay? And that's the way you actually feel about it, and that's the much more functional way of actually dealing with everything. Okay? The fact that recursion's involved, that has nothing to do with the functional paradigm, really. That has more to do with the fact that the central data structure in the language is the list, which is a recursive data structure. Okay? That make sense to people? Okay. As far as Eval is concerned, when would you use this? It requires a lot more – the examples are much more complicated. Where I've seen Eval used is, I've seen Eval used where Apply couldn't be used. You could imagine applying a plus, or a cons, or a char, or something like that, to arguments. Right?

You can't Apply define, or add, or or because they're special functions that don't necessarily evaluate all of their arguments. You understand what I mean when I say that? Like and, and or short-circuit evaluate, right, so they actually cannot be thought of as normal functions. If you wanted to "Apply," and I have to use it in quotes, "Apply" and to a list of predicates, to figure out whether or not they're all true or not, you couldn't use Apply because it really requires this to be a bona fide function, and and is not one of those. But you could cons onto the front of the list and then evaluate it. Okay? Does that make sense? Okay. I've also seen—and this is really sophisticated stuff – I've actually never coded this myself, but I have seen, programmatically, the definition of new functions have been synthesized textually. Okay? Do you understand what I mean when I say that? Like, you actually can build up, as a string, something that looks like the definition of a function. Okay? And I say – I meant a definition of a function, something like this. So imagine an algorithm, even though we don't an example of this here because I think it requires a really sophisticated setting, in order for you to need this. But you could imagine this being built up as a string in a language. Okay? Or built up as a list, rather. So if, somehow, you can textually synthesize that right there, and use cons, and char, and cuda to construct all of these symbols in one big, complicated list and then, pass all of that to an Eval statement, you can programmatically introduce the definition of new functions while the program is running. Okay? That's actually a pretty neat idea; some people would say it's dangerous that a code would be this self-aware and evolving while it's running. Some people were like, "I don't – that's fine as long as I'm a good enough programmer and I can handle it. Then I'll just lever it like – be able to use that feature of the programming language."

I've never seen it, but I have heard enough, and read enough about the contribution of Eval and the introduction of evolving functions, and functions on the fly, in things that involve randomization and things like genetic algorithms, where new sentient beings are modeled, you know, while the program is running. And they have different little definitions of how they should execute and respond to the world, in the simulation. Okay? Does that make sense? Okay. Now, I'm gonna focus on this one because I think it has a broader impact on the type of code we're gonna write. What I want to do now is, I want to revisit the flatten problem, now that I know about Map and I also know about this thing called Apply. Okay? Now, I'm gonna revisit the flatten thing. That's the char of sum list I want to flatten. Here is the cuda – I'm sorry, here is another list, another element that I want to be involved in the flattening process. And then, I'll have just 10, like that. And you know what the product of – you know what the product of the flattening of that should turn out to be. It should be 1 2 3 4 5 10, all as peer top-level elements and no intervening parentheses. Parentheses as bookends, okay, but nothing in between, other than the numbers. Does that make sense? When we implemented this, first we recursively generated the flattening of this, and then we either consd or appended, or prepended, this element right here onto the front of it. Okay? In this case, we would flatten this. It wouldn't be very hard, but we would just flatten it. It would be very quick about it. And then we appended this to that right there. Does that make sense? Okay? If this had been an atom, then our [inaudible] list that we just would've cons this onto the front of the recursive flattening. Well, what I'd like to do is say, I could actually – while I'm defining this, this is kind of kooky but, while I'm defining this flatten function, I could implement it recursively. I could recursively flatten all of the elements, where this gets transformed into a 1 2; this gets transformed into a 3 4 5, and this gets transformed into a 10, just temporarily. Okay? Do you understand how the definition of flatten could involve a recursive call to flatten, but not directly, but via Map?

Oh, I want to flatten the entire thing. Oh, I should Map flatten over the list, okay, and then append all of the things that I get back. Does that make sense to people? Okay? If this is the product of flattening the first element; this is the product of flattening the second element, and I just ensure that this is the product of flattening the last element. Okay? Then, I could effectively – I could effectively get the final product by just appending all of these lists. I could Apply – this is gonna come back like that, if I really use a Map call. Okay? I could Apply the append function to the product of the recursive Map call, so that it goes through with a thread needle and just, basically, creates one big sequence out of all of these elements right there. Okay? It's really kind of hot. So let's implement this, okay, and we'll use some leap of faith arguments to defend why we know it's working. But what I want to do is, I want to define the flatten function and I'm just gonna give it a sequence. Okay? Let's just – let's forget about the base case. Okay? We're not even sure what the base case is. Okay? Here, actually, is a base case. But let me just think about the recursive case, where if I know I'm dealing with a top-level list of many items, then what I want to do is, I want to Map this flatten routine, that's being defined right here, over sequence. Now, think about what that does from a leap of faith standpoint. This transforms a list – this list of length 3, okay, into this list of length 3. Okay? If flatten works, in this, like basically, in this involuted way where it actually – the definition of flatten is compatible with itself. Okay? It's supposed to transform this into this, right here, via this one Map call. Okay?

And after that happens, I can Apply, not plus, but I can Apply the append function to the result of that Mapping. Okay? It's like taking the list of length 3, pulling the left parenthesis in a little bit, and sticking the word append at the front, and say, "Okay, evaluate yourself as if append were there all along." So if I stick in, right there, the word append and evaluate it – and that's what the Apply statement there is doing – it will give me one list with 1 2 3 4 5 10 in it. Okay? So that's fun, I think, but we, actually we only want to do that if this thing really is a list. Okay? If it is the case that the sequence itself is not a list, okay – in other words, if I actually, recursively, hand a 10 to this thing – does that make sense? Okay. Then I want to just return the listification of this sequence. Okay. Otherwise, I want this to be the else flaws; that balances that – I'm sorry, that balances that, balances that. This ends the if, and this ends the define. Okay? So the base case deals with the scenario, when you dive so deeply into the recursion that you've actually arrived at a 1, or a 2, or a 3, or a 5, or a 10. Okay? And you say, "Okay, now I have to start backing up." But because cons append is involved, I have to wrap them in parentheses, so that they behave nicely in the context of the Apply append call. Does that make sense? Okay? Think about what happens – this is recursively flattened, according to that formula. Okay? It just works out. Same thing with this right here. When it recursively calls flatten against the third element in the sequence, it gets this atom, which is not a list, so it has to wrap these things in parentheses so that, when it participates in the Apply append call with all of the other peers that are progenerated recursively, it actually plays nice. Okay? You guys getting this? Okay, very good. Now, some people do not like the fact that I gratuitously put these parentheses around the elements, all of them, just for them to disappear. But if I really just want to illustrate how Apply and Map are working together, then this is still a good vehicle for this function, I think. There's no very easy way for you to look at a list, and know whether all the things inside of them are atoms. So the list 1 2 3 is different than the list 1 2 list 3. Right? Okay? If I could, somehow, detect that 1 2 3, everything inside is a top level atom, then I could come up with a more sophisticated implementation here, that's a little bit faster

But all I'm trying to illustrate is how Apply and Map will end in this one example here. Okay? That make sense? Okay. This is probably the tightest recursion – example of recursion you may have seen, ever. Okay? In C + +, it's buffered with all of this memory allocation. Okay? And it's not so clever in its use of base cases. Scheme is this very expressive – that's the positive PR spin on it – it's this very dense way of expressing algorithms. It's usually as terse as the most articulate person is in describing what the algorithm's supposed to do. Okay? And that is very difficult to look at and understand how it works. Okay? You guys are good? Okay. I have one other topic I want to introduce, and I'll spend a lot of time, on Wednesday, going over very advanced examples of all of this stuff. But Scheme has been different than everything we've seen before, in the sense that it's almost entirely runtime; there's no compile time element to it, whatsoever. It is weakly typed, which doesn't mean that there aren't types involved; it means that all kinds of type checking is deferred until runtime. Okay? And if there are problems, then it just presents the problems, if it ever comes up, while the code is running. There's one next thing, actually exists in extensions to C and C + +, but it's not part of the core language. I want to think about how we would implement this function. I want to define a function called translate. And I don't mean translate in a linguistic sense; I mean translate in a distance sense. I want to take a list of points in one-dimensional space – so like points on the number line – and I want to shift all of them by a certain delta. Okay? So I'll just say something called points, and I'll say delta, right here. Now, before I go in and fill in the body here, this is how I want it to work. I want to be able to call translate against 2 5 8 11 25, and I want to be able to pass in 100. Okay? And I want it to spit out 102, 105, 108, 111, 125, just like that. Okay? I want it to take the lists – the first argument that's a list of length n – n points, and I want it to generate another list of n points, where everything's been shifted by some delta amount. Okay? Does that make sense? So you look at this and, given that I just taught you Map, 35 minutes ago, you may say, "Okay, well, maybe he wants me to use Map." And the answer is, I do want you to use Map. Okay? But, the problem is that, there's no clear existing function that knows how to translate a number by another number that's not specified, until the function call actually happens. You understand what I mean when that feels a little bit like client data to this Map call right here? It's external to the actual thing being Mapped over, but it, somehow, is involved in the product? Does that make sense? So I kind of want to Map something, right there. I'll just put this big placeholder, and that's supposed to be the function that somehow figures out how to add this number to every single element inside points.

I have a very difficult time naming a function right there because I can't call a function called increment by delta because that type of function is gonna have to take two arguments, not just one. It will have to take one element that gets bound to the char of the list that it's Mapping over, and you'd have to also pass in the 100 to it. Does that make sense? Okay? You guys understand the problem here? Okay? This has to, basically, either be global data, okay, or it has to behave as global data in the execution of whatever function gets placed right here. Now, I'm moving a lot of space for this thing right here. What you can do in Scheme, and a lot of other languages, but not C or C + + is, you can actually embed the definition, and scope the definition of a function, inside another function. Okay? This is the way you do that: You erase the placeholder boundary. There's that. You can actually define a function in Scheme, on the fly, using a keyword called Lambda. And Lambda is just a gesture to the calculus that backs most programming languages, and the way it deals with function call and return. But if I write this right here—just think of it as boilerplate – that means that I am defining a function without giving it a name. Lambda is not the name. Lambda is just a placeholder, meaning it's an anonymous function I'm defining on the fly. Okay? I am saying, quite clearly, that this anonymous function takes one argument. Why? Because the way we're using Map, right here, we're Mapping over a single list right here, so I need to actually let every single thing that is ever a char of a cuda, okay, actually be passed in and bound to x. The body of this function has to add x to something. It has to take the 2 to a 102, or the 5 to a 105, or the 11 to a 111. Okay? The only way it's gonna do that is if its definition can involve the actual value that delta has adopted on this particular call. Okay? Does that sit well with everybody?

So I can do this, and that ends the Lambda definition. So this, right here, is this anonymous function, whose implementation is framed in terms of the one parameter called x, and something that is effectively global to its scope, called delta. It happens to only exist as a local variable to the outer function, okay, but for the lifetime of this function definition, it exists only long enough for it to be Mapped – for it to be Mapped over this thing called points. This is going to adopt whatever value was served the actual translate call. Okay? So that, if I call this with 100, it constructs an anonymous function, on the fly, where this is the number 100. If I pass in and call this thing a second time, but I put 1,000 there, it constructs a second function, okay, that is, from a memory standpoint, is completely independent of the first indication of this. And it actually just puts this – places a 1,000 there as opposed to a 100. Does that make sense? Okay? So that is how that feature works. It is not anything you've seen in standard C or C + +. Now, it turns out that G + +, which is all about extending C + + because it just doesn't like the original language, I guess. It allows you to define inner functions. I haven't advertised it to you. In fact, I didn't know it until a year ago, when some student showed it to me. I'm like, "Uh, let's pretend that that doesn't exist because I don't want people using it." But this, right here, is actually quite common, in the Scheme world, anyway. Okay? There are two things I want to announce in the last four minutes before I leave you go. You actually can explicitly define inner functions, if you want to. I actually like both ways. I think this one, it's kind of jockier. It's like, "Yeah, I'm not intimidated by – I don't need function names." is kind of what it's saying. "I don't need to wear my seatbelt when I'm coding."

So. But there is a more expressive way of doing it, that I think is fine. If I wanted to define this translate thing, a little bit more – I don't want to say eloquently – it's just – I guess so, a little bit more clearly and be more obtuse about the fact that some inner function is involved, I could define translate of – I'll just call it seq and delta, just like that. I could actually provide an inner definition – oops, keep making this mistake – I can define this inner function, internally, called shift by. And I can actually give it a name where I actually add delta, okay, and then, right afterwards, I can Map this shift by function over the sequence, and that ends the entire define. Now, the reason I don't like this is because it kind of breaks the functional paradigm. I've always equated the definition of some function with one expression. Right? I've – increment is equated with the plus function; translate over here is equated with the Map function. There's always been one expression that was provided as the body of the function, whereas I'm actually providing a sequence here. This is like, Roman numeral No. 1; this is Roman numeral No. 2. Pure Scheme allows you to actually define a sequence of items, okay, and then the expression of whatever the last one evaluates to is what the overall function evaluates to. I don't like this because it isn't purely functional. All of a sudden, it takes on this C or C

+ + like idiom, where it constructs a lot of things piece-meal in order to build up the overall answer. Okay? But you can define an inner function, if you want to; materialize a meaningful name for the short term; define it in terms of local variables in – global symbols and local variables plus n delta, and then use the last line to actually define what the Map should do, or what function should be Mapped over the sequence. Okay? Does that make sense? Okay.

So there's one other thing I should tell you about the define thing. When you write something like this: Sum x and y, and you just – a simple placeholder definition, just to illustrate my point, right here – that right there, is just syntactic sugar for this. Define the sum to be equated with – now, you may be a little weirded out by the syntax, but the second one is actually much more clear about its association of some function with an actual name. Okay? Does that make sense to people? So that, every time you use sum in a call, it evaluates to this Lambda function that's compatible with two additional arguments. And then, this executes where x and y have been replaced by whatever those two additional arguments evaluate to. Okay? Define actually works – if I wanted to do this – actually, I shouldn't use x. Let's say I want to do, like, PI 3.14; you haven't seen this before and I don't want you to use it, but you understand what's probably happening there. It's forever associating capital P, capital I with the number 3.14. This is just, basically, doing the same thing. It just happens to be associating the word sum or the symbol sum, not with a constant – I'm sorry – not with a numeric constant or a string constant, but with a Lambda constant. Okay?

And so every time you use sum, or char, or cuda, or append, or Map, or my unary Map, or whatever, the actual symbol that's at the front, sum plus even, technically – plus and minus and times and all of those – they're all bound to actual Lambda functions. We prefer this because it's probably just a little bit more readable, and it's more consistent with the way we've learned how to program in other languages. But this is much more clear; I want this, but this is functionally equivalent. This advertises quite clearly that sum is being equated with this thing, right here. Okay? Does that make sense? Okay. So Scheme really is all about symbol and symbol evaluation, and functional evaluation, and right down to the actual definition of the functions, and the way they're stored in memory. Okay. So I will cover some more sophisticated examples on Wednesday. You'll also see some sophisticated examples in tomorrow's section, as well. So again, I –

[End of Audio]

Duration: 50 minutes

ProgrammingParadigms-Lecture22

Instructor (Jerry Cain):Everyone, welcome. I have no handouts for you today. I have several handouts from the past days that I have not finished yet and, in fact, if there's one handout that I consider to be the advanced scheme handout it's probably handout 32. I'm gonna be going through two of those three problems that are in there right now. You saw some sophisticated scheme examples in yesterday's section handout. I want to do a few more today. In particular, I want to get this lambda function idea down and out and some significant examples. I only went through a very trivial example of how lambda gets used last time, but I really want to do some complex, dense, recursive scheme functional programming using mapped lambdas today.

The first one I want to do is actually more of a discreet math problem than it is a scheme problem, but we're gonna drive the problem using scheme. It deals with the notion of what is called a power set and those in 103a and 103b over the past year are very familiar with the power sets. The loose definition of a power set is that it is a set that contains all of a sets subsets. So if we talk about the original set just {1,2,3}, but you know what, I'm not gonna draw it that way because we're not gonna draw it that way in scheme. The set {1,2,3}. Okay.

If I want to list every single subset of {1,2,3} I have to basically enumerate all the subsets such that either one can be excluded or included, two can be excluded or included. There's two options for the presence or absence of every single element. So for this particular set right there there are going to be a total of eight subsets. So the actual grouping, they won't necessarily be in this order, but I somehow want my power set function to be able to eat and digest this as a list and synthesize something like this. The order will be a little bit different, but I will do it this way. Two, three, I'm not sure this is the best order, but there's a method to my madness here.

Do you understand how those four – they're all technically subsets of this thing right here. There's no element in any one of those four sets that I've enumerated that can take the element that isn't there. So technically they're all subsets, including the empty set. Okay. Here are the other ones. And that's the full power set expressed using just scheme lists. Okay? Now, the reason I drew it this way is I'm trying to make it clear what recursive structure I'm gonna exploit by my implementation. We all agree that those are all the subsets that exclude the one. Okay? Does that make sense?

These are all the subsets that include the one and I've ordered them in such a way that there's a clear 1:1 relationship between one subset and the one below it. Each one of these four things right there is identical to those except that a one has been prepended as the English word cons as a scheme word to the front of the list. Does that make sense to people? Okay. So it's almost as if I want to recursively generate this right here. I want to recursively generate it again, but the second one I want to cons a one onto the front of it. Basically, I want to map some function over this list right here that knows how to cons a one onto the front of it. Is that so odd everybody?

And then I want to append this list that has no one's whatsoever to the list that has all ones in it. Okay? As far as the – I'll draw it this way. The power set of the empty list is concerned you may think that this generates the empty list. That's technically not true. That would speak if there are absolutely no subsets of the empty set and that's not true. Technically the empty set is its own subset. Okay? It's called an improper subset, but it's technically a subset. So I expect my power set function when given this right here to return that. Okay? So it's a singleton that has its own element, the empty set. Okay?

I always expect the lengths of these lists to be a perfect power of two. So I have two to the zero for this one right here. That's the length. Two to the eighth as this right here. Okay? It's very clever. It's very dense. I'm gonna write this twice. The first time I'm gonna be careless in making the same recursive call twice. Okay? But I want to go ahead and define – I'm just gonna call it PS. Not postscript, it's short for power set. I'm just gonna give – name the variable set. That's not a key word we'll pretend it's not a key word in any scheme dialect. I don't think it's one in ours, but just pretend that I can use that as a variable name without interfering with the name of some built in function.

I just want to do a simple base case check right up front. If it's the case that I have the null set it's like the language is working in our favor. It's awesome. Then according to this specification right here I want that to prompt the entire thing to evaluate to that as a constant. Okay? That technically is incorrect. Okay? That would go in and pull out the empty list and assume it's bound to functionality because of the place it's occupying in the list, but by putting the quote in front that suppresses evaluation. Okay? Makes sense?

Otherwise what needs to happen is that I have to somehow encode the appending of this right here to the appending of that right there. This right here is just the power set of the cdr of a set. Forget that the one is in there. Take the leap of faith. What is power set of {2,3} supposed to generate? It's supposed to generate that. Okay? It's just best to get this on the board and to just show you – explain why the code is correct. I do want to append two things. Okay? I certainly want to generate those two lists and then concatenate them. The first one I get by pure leap of faith of calling power set of cdr of the set. You're just assuming while you're writing this, of course, that the PS function works while you're writing it. That's just the leap of faith principle.

The second thing is actually a little bit more involved. The second half of the entire power set is certainly related to the recursive call, but we somehow have to visit every single subset in the recursively generated power set and cons the missing element onto the front. We were correct to omit the car from anything that had to do with this power set right here. This is the first 50 percent. That's all about excluding the one or the car of the original set. I want to append this to more or less the same thing. I'll call it subset. Where I cons the car of the set onto the front of a subset. Now, that is not over with. Let me just do PS of set. That ends the map. That ends the append. That ends the if. That ends the define.

Let me do you a favor and make it clear what function is being mapped over this power set. I'm sorry. That's not power set of set. It is – that was not quite right. Cdr of set ends the map append if defined. This is the on fly function that I told you about on Monday. Okay? It is interesting because I really am defining it as part of the execution of power set and its implementation is framed. This is the first time we've formally seen this as part of a language. Its implementation is framed in terms of not only this intervariable, but the value of a local variable in the outer scope. Okay? Does that make sense? Okay? Question in the back? Just pencil? Okay.

So I'm mapping. This is a function over whatever the recursively generated power set is. Okay? This accommodates the issue, or deals with the issue, that the one or the car is excluded from the recursively generated power set that says you know what? I'm gonna visit every single subset. If I actually apply some functionality to every single thing in the recursively generated power set every single one of those things is one of a legitimate subsets of the original. The map function visits each subset in the recursively generated power set and it cons's an element onto the front of it.

So it transforms each subset into another subset that has, as opposed to excludes, the car. Okay? Does that make sense to people? Okay. This is all of it. Okay? You can look at that and it's very easy for you to convince yourself that it works because it has all the right pieces and because the lecturer's writing it and you just usually trust him to write the correct code on the board, but it is very difficult to write the very first time you do it from scratch. Okay? It's just very difficult to basically figure out how to invent the lambda and how to get the append call as opposed to cons call to do the right thing for you.

So even though that looks compact and easy and articulates the algorithm pretty nicely it's very difficult to come up with from scratch. So make sure you read through the first page and a half of Handout 32. I went to great pains to try and make sure that you understood in this literary way how we were constructing the power set, so that you can revisit the descriptions in case you ever have to rewrite something like that from scratch. Okay? There is an issue with this. It's not a huge issue from an algorithmic standpoint. It is functionally accurate and functionally correct. It unnecessarily takes an exponential amount of time. Okay?

I'm sorry. I shouldn't say it that way. It takes an exponential amount of time anyway. It actually makes the same recursive call twice. Okay? And so this is the best example I have to introduce this construct that was introduced in an earlier handout, but I just haven't talked about yet. But that is a call right there and that is a call right there. They're exactly the same things. Set is not being updated or being altered by reference in a way you could with ampersands and asterisks from C and C++. All the lists in scheme are these immutable things and given the constructs that we've learned you don't actually modify existing lists. You synthesize new lists out of old ones. Okay?

That's what cons is doing and that's what car and cdr are doing. Okay? What I want to do is I want to figure out whether or not it's possible to just call this thing once and use whatever it evaluates to into two different settings. And the answer is yes or else I wouldn't be talking about it. Okay? There is this construct in scheme, right? It's called a let binding. It is actually very similar to a lambda and, in fact, I'll explain what a let really is. It's really just intactive sugar for calling an inner function. Okay? This is functionally correct. It runs very, very slowly for any set with more than, like, say six or seven elements. This is more intelligent.

Define power set of set. The base case is precisely the same. Null set. Go ahead and return that right there. Otherwise, what I want to do is I want to execute some block of code. Before I execute the block of code I want to evaluate an argument. Okay? You use what's called a let binding. It's basically saying please let this variable equal to whatever this evident expression evaluates to. I only have one let binding here. I'm going to bind PS rest to whatever I get by calling PS against the cdr of a set. That is the cdr. That ends the P set. That ends the pairing. That ends the full let statement. So this right here is balanced by that. This right here is balanced by that right there.

Formally, this is everything between this parentheses and that parentheses is supposed to be a list of pairs. I only happen to have one pair that's necessary for this, but if I had several variables I wanted to initialize and use in what I'm about to write then I could just provide a list of them. There's actually a clear boilerplate in one of your earlier handouts as to how to script that out. Okay? But what I'm basically asking is I'm asking scheme to let me associate this as a variable name. Much like that and that are. Okay? I'm sorry. Much like set is. And just associating it as a single symbol with whatever this evaluates to. Make sense?

It's more than just being clever and only calling this thing once. It actually saves a huge amount of time because now what can happen is I can append PS rest. Functionally exactly the same thing as that right there to whatever I get by mapping the lambda over a subset. The lambda doesn't change. Cons, car, set, onto the front of the subset. That ends the cons. That ends the lambda. And I can map that over PS rest. That ends the map. That ends the append. The let is the define. Okay? Does that make sense to people?

This right here is the let finding. This parentheses doesn't close the let. It closes that one right there. Everything right here up through the paren that, this one right here, is under the jurisdiction of the let statement. So not only does it have set as a variable, but it has this thing called PS arrow rest as well. Okay? That's been associated with the recursive call. I only have to make the recursive call once. I can remember the answer. Okay? I can remember the answer in a code block right here and it saves on running time considerably. Okay? If you have basically the structure of a let statement, basically I'll just make some things up. Like X expression one, Y expression two, Z expression three. This is just me inventing a syntax for making it clear what the let's supposed to look like.

This right there closes that right there. This is either one, usually one, but technically could be a series of expressions that are evaluated under the jurisdiction of the let, which means that you can refer to X, Y, and Z and whatever they've been associated with. So this is just some functionality of X, Y, and Z and whatever other variables were available to you. The point I want to make is that you may look at this and say, okay, the first thing it does is it evaluates the expression one and it associates it with X and then next in this very sequential way it evaluates this and buys it to Y and evaluates this and buys it to Z and then carries on to this.

Scheme doesn't actually pledge to do it that way. For reasons that will become clear in a second, you cannot assume anything about the order in which those expressions are evaluated and the order in which the variables are bound. You may say, well, does it really matter? And the answer is yes because you may think because of the way you write this that you can refer to X in this expression right here and you can refer to X and Y in that expression right there and you cannot. You have to think about these three things or those N things as being evaluated either in parallel or in whatever order you want. Okay? Does that make sense to people?

The scheme interpreter is free to do whatever it wants to. If you really do want to stipulate that they be executed and the variables be bound in the order that you prescribe them, you have to use a variation on let called let*. The asterisk is this very abusive symbol, which like stares at you and says you're using a piece of a language I don't want you to use. Okay? But let* does impose a sequential ordering in the way that things are evaluated. Okay? To the extent that you use let* you're shifting paradigms. If you go from this purely functional thing where everything is the composition of some simpler function. Okay? Then you're being purely functional about it.

When you do this, all you're really doing is C programming where you set X equal to an expression and Y equal to some expression that involves X. Okay? Does that make sense? So this with the let* right there, which I don't think you have any reason to use in assignment six or seven for that matter, but I'm just talking about it for completeness. There is no compelling reason to use that unless it just doesn't make sense for you to make repeated function calls and you really do depend on the order in which things are evaluated. I will erase that right there. As far as let is concerned I actually don't know how it works behind the scenes, but I can give you an idea as to how it more or less works. What theory guides the let construct.

You've probably all read a little bit about let in the handout. Maybe you haven't because assignment six was due two nights ago and scheme isn't due for another week, so you may just be taking a vacation. But let me just explain what let is more or less equivalent to. When you do this let, and I'll just write it this way, X something, Y something, and then you write some block of code. A of X and Y. Do you understand that in some ways it's almost like a function call? It's like you're evaluating this argument right here and associate it with a variable called X. You're evaluating this and associating it with a variable called Y. You're executing this as if it's the body of a function. Okay?

This is a more sophisticated clever use of lambda, but I can totally explain what this is more or less compiled to or translated to after it's been read by the scheme interpreter. I've just opened a paren. The thing that normally comes after an open paren is the name of some function. Okay? What I'm gonna do is I'm just gonna put lambda – you're used to putting symbols there that evaluate the lambda's. Okay? They're bound to cope, but you can put the explicit code at the front if you want to. I'm gonna put lambda of X and

Y that calls A of X and Y as part of its implementation. This is the entire car of that list. Okay?

And its body of that lambda function is more or less synonymous with the body of the let statement. Does that make sense? Right here is the first argument to this thing. This thing expects two arguments. I can just put whatever expression one is and expression two is and then call it a day. So what the let thing really ends up being is just a rearrangement or a different way of expressing the application of some anonymous function to the expressions that you're evaluating at the top. Okay? Do you understand what I mean when I say that?

If you look at the let it makes it look like the evaluation of those arguments is happening first and then some code block is executed under the context of those variable assignments. That's exactly what happens with function evaluation, too. You evaluate the parameters. Okay? In whatever order that they want to and scheme doesn't dictate what order the parameters will be evaluated in. So this gets evaluated, this gets evaluated. You certainly do not want to have the result of this thing right here influence the evaluation of that or vice versa, right?

That's a product of the rule that you can't rely on the order in which these are evaluated. Okay? It's unusual for you to see this at the front. You can do it. It's syntactically correct and valid to do it that way. Okay? But this is functionally identical to this and any limitations that I'm imposing just by rule of this right here about what order things are evaluated in can also be said about this. Does that make sense? Now, except for I have a couple other scheme functions I want to write, but except for the fact that there are a library of scheme functions. Like conceptually you know, like, 60 percent of the language all ready.

Scheme is famous for a variety of reasons, but the one I'm thinking about right now is that syntactically the language is very, very easy to learn very, very quickly. Now, it helps that you all know Java and C and C++ very well now. So you can always equate something that looks confusing with some equivalent or near equivalent in C or C++ or Java and so it's easier to learn your fourth language than it is to learn your first one. But there are many curriculum's that until recently, and even today, have decided that they'd rather teach scheme in their introductory classes. For 20, 22 years MIT taught scheme in its introductory computer science class. Cornell still does as far as I know. So does Berkley. They all use the same model. MIT has just recently migrated away from it.

They're in transition right now to teach Python as the first language. Their argument is that there's no exposed dynamic memory allocation. There's no freeing the leading asterisks, ampersands, all of that arrow nonsense. There are classes in some extensions of scheme. There are structs as well. We're not using structs; we're using lists for everything. It's like, oh, you destruct while you use the list and make sure you know that the zero element is always the name and the first element is always the GPA or whatever. Okay? And their argument is that the language is very economical. It's terse, it's expressive. Expressive is good PR term for dense. Okay?

It forces people to think about their abstractions and about their algorithms a lot more quickly than they do in C or C++ where they have to be worried about variable declarations and memory allocation and things like that. Okay? Does that make sense to people? That's the first time I've explained the let thing because I actually never knew what let was equivalent to until last quarter when somebody told me about it. I'm like, ah, I can probably talk about that in 107. Okay? So this explains the theory behind let. That's a legitimate example of where you'd want to use let. Okay? Basically you evaluate that thing right there and that big append is like a lambda that takes one argument. Okay?

Its argument is whatever power set of cdr of set evaluates to. Okay? Just use this as the analog for that. Okay? Now I have a more difficult mapping of a lambda problem. This is a single mapping. Okay? We map lambda over this and it's confusing in its own right, but as far as mappings go it's a standard mapping where the function you're mapping over happens to be a little more complicated than it would normally be. I have this great example. It's another common [inaudible] number theory thing like you would see in 106, 103b or 103x. I want to write a function called permute. I have to assume that 90 percent of you have written this in C or C++ because you all went through 106b or 106x here, but I want this as a function to output all, in this case, six permutations of those numbers.

I'm gonna assume that all of the lists are simple lists. There's no nested list inside the list and that there's no duplicates and I'm just gonna assume its order from one through N. Okay? And not worry about duplicates or anything like that. It'll still work even if I don't do that, but I just want to deal with 1, 2, and 3 or 1 through N. This is supposed to output this. 3, 1, 2. That's 3, 2, 1. Okay? I'm gonna have to do a larger example of this. Just a sample function call in a second. The way I wrote this it's technically illustrating the structure that I want to exploit in my algorithm.

These right here are all the permutations that begin with the number one. These right here are all the permutations that begin with the number two. These are all the ones that begin with the number three. Okay? Without going into the scheme code, if I'm interested in the permutations of 1, 2, 3, 4 I'll just write the ones that have one at the front. 1, 2, 3, 4. 1, 2, 4, 3. 1, 3, 2, 4. 1, 3, 4, 2. 1, 4, 2, 3. 1, 4, 3, 2. Then they're all those. Draw a little box around this. Then there are all those that begin with the two. Then there are all those that begin with the three. Then there are all those that begin with the four. Okay?

There are 24 permutations here. It's not 24 so much as it is four factorial. These right here, like little Russian dolls of permutations. That happens to be, what I've just boxed with the inner rectangle are, all the permutations of the cdr. Okay? However, I'm not gonna draw them in here, but what's in there isn't the permutations of the cdr. It's the permutation of whatever you get by removing this element from the list. These are the permutations of 1, 3, 4. These are all the permutations of 1, 2, 4 and 1, 2, 3. So when I say that this is a permutation of the cdr it's not the best way to say it. It's the permutations of the original list with the one removed. Does that make sense? Okay.

Now, I'm just gonna assume that the remove function is a built in. It actually is, but it's not called remove. I actually wrote it in the handout as if it really is just a built in. I'm sorry. Not a built in. That it is in a built in and I had to write it myself. You may think that mapping has nothing to do with this and it would be a completely reasonable thing to say because we're taking a list of length three and transforming it into a list of length three factorial. We're taking a list of length four and transforming it into a list of four factorial. That make sense? But what I'm gonna try and do and I think I can do it because I've written the code now 17 times.

But you can actually do this because you can use mapping in a way that you're not used to, but I can actually make the one responsible for somehow transforming itself into all of the permutations that begin with the one. Okay? With two transforming itself via mapping, clever mapping, but mapping, into the list of all those things that begin with a two. Does that make sense? So the way I'm gonna draw this is I'm going to rely on mapping to take something like 1, 2, 3 and transform it into the list. I'll call it one perms. I'll call this two perms. And I'll call this three perms.

When I talk about three perms I'm talking about all the permutations of this list right here that begin with a three. Okay? And I want this to expand to that. Whatever mapping function is applied to this list right here has to transform this and this and three into those things right there. Now, I don't want the permutations to be subgrouped with extra parentheses based on what element they begin with. The ultimate answer is I don't want these parentheses. I want all of the things, whether they're one permutation or two permutations or three permutations, to be here. Okay? Make sense?

What I want to do is I want to recognize based on what I've said right here that the overall algorithm for permutations without worrying about base case yet. Might as well just erase with the chalk. I want to take define and I'll call it permute and when I write this thing right here I'll just call it sequence. I really am thinking and framing it in terms of sequence for sequences like 1, 2, 3 or 1, 2, 3, 4. Okay? I basically want to map some function to be determined and by drawing it the way I do you should just know that it's going to be a lambda. Okay? And I want to map that over sequence. In fact, I'm gonna change the word because now I remember what word I used in the handout. Items.

We have no idea what this is, but if we know that something can be put there and we can make it work the thing that makes it work is gonna take this and transform it into that. We can't have those intervening parentheses, but what we could do is we could – I love this, you can't do this in many languages very easily. Okay? I'm given M lists. I want to take those M lists and I want to take an append and I want to put it onto the front and then evaluate it as if the append were at the beginning of them all along. That flattens it to one degree. Make sense? Okay.

Now, unfortunately, this is not easy right here. Okay? We have a mapping function. You know what's going on here, but we have to somehow take a single element out and I'll admit right now that it's gonna be some lambda. I'll say that it's either gonna be a one or a two or a three. I have to somehow take a lam, which is a one and transform it into a list of all the permutations that have one at the front. Does that make sense? Now, this is a run on sentence I'm about to get to, but let me try it. When this thing is one, I have to have the function that's right here. Somehow transform the one into all the permutations with one at the front. I get that by mapping another lambda over all the permutations of the set where one has been removed.

What is the mapping function? It's just like it is for the power set where I cons whatever this element is onto the front of all of the permutations that happen to exclude this element. Okay? That make sense what's going on? Okay. Now, I'm concerned that I don't have enough space. I'm just gonna remember that item has to be drawn at some point. This is the lambda and I'll get to the items at the end. I have items outstanding and I have a base case to worry about right here. What do I do on behalf of each element?

I actually want to map another name. I'm not gonna commit to a name yet. Whatever I get by calling permute of remove of items LM. That ends the remove. That ends the permute. That ends the map. That ends the entire lambda. Okay? There's so many pieces here and I know it's confusing, but I think you're gonna be able to get it. Okay? This right here is just some function that figures out how to get a one into all of the permutations where one is at the front. Okay? In order to do that I have to recursively generate all of the permutations that you get by excluding the one from item. There's a leap of faith argument right here. Okay? Does that make sense?

What function gets applied over these things? Well, these things right here are actually permutations, so I'll call that argument permutation because I'm applying some function to each permutation in the permutation set of the set that excludes one. Okay? What do I put here? I cons a lam onto the front of the permutation. That ends that. That ends the lambda. This was the argument of the map call. That ends the map. That ends the map. That ends the map. I don't want this one right here. We'll just be clear about which ones really need to be written in at the moment. This ends the remove call. This ends the recursive permute call. This ends the inner map. This ends this lambda right here. Okay?

This map has to map over something. It has to map over the thing that actually gives me isolated elements. This is where items go. And that ends the apply call. Okay? Make sense? Now, the one thing I'm not doing here is that this thing is infinitely recursive at the moment. It's gonna eventually try to take the clear of the empty set unless I block it. Up front all I'm gonna do is if I have an end to items list with the same reason as before I'm gonna return that. I'm gonna assume that the empty list has the empty permutation as its only permutation. Zero factorial is one. Okay? So that's why I have a list of one as opposed to a list of zero there. Okay? Does that make sense to people? Okay.

To look at that it could be just, like, morally offended by how dense that recursion is. If you have a double mapping with a double lambda it's really gonna force you to stretch to figure out how this is accomplishing the task and where the actual permutations are being constructed. This cons, this is really the only dynamic memory allocation function that we really talk about. This, an append, are the things that build larger lists out of smaller ones. Okay? Behind the scenes any time cos is invoked that's really a request to build some linked list cell. Okay? That didn't otherwise exist. And it populates the two fields of the linked list cell with that and that. We'll see next time that's exactly how it works from a memory standpoint.

But it's the accumulation of all of these cos calls and this apply append call where it actually extends permutations and merges lists permutations as the recursion on lines. Okay? And it bottoms out when it does this right here. Okay? Make sense? Okay. Very good. If you're getting this then you are – this is the hardest part of scheme. It has very little to do with the language. You understand lambdas. You understand mapping. It's actually combining them with their expressiveness/density to actually get algorithms like this to run really cleanly. You can do this in C. You can do this in C++. You manually manage the swaps. You've written permute, I'm sure, on strings in C++ in 106b or 106x. Okay?

The string class shields you from some of the memory allocation, but schemes win even over C++ and C is that it obscures all of the memory details from you entirely because the list is this central data structure in a way that it's not in C or C++. Okay? So you have built in support for list dissection and extension. Okay? You guys doing okay? Questions? Okay. Also explained in the handout. Okay? This was an old exam question. So were the things that were on the section handout yesterday. This was intended to be the hard scheme question when I gave it. Okay? This was like nine years ago. I'm running out of scheme questions, so you may see it again.

This is really the densest thing to do and it's hard to look at, but scheme does it much more expressively and better, I think, than C and C++ would. What I want to spend today on and at least half of Friday, if not all of Friday, is talking about the memory model that's involved in implementing scheme. I've made a couple of points, but these are takeaway points that are good to remember even if you don't continue programming. Scheme functional paradigm. You don't think in terms of variable assignments or outline form. You don't think about objects. You think about the data, but not really. You just think about functional languages like scheme and ML's another example of one.

There's actually a language at the moment that's all the rage called a Haskell. I might have a coworker of mine come in and talk about Haskell during deadline week. It's about data transformation in this very functional algebraic way, but it's algebra that's extended to include lists and strings and Booleans. Okay? When you program in scheme you program without side effects, or at least you intend to. Certainly you do in the subset of scheme that I've taught you. That means that you rarely rely on your ability to update a local variable and somehow expect that to be reflected in the argument that was passed from the color. Okay?

You can do it. There's an example in the handout. The partition function that I wrote to help implement all of quick sort uses some sort of clever way of actually programming with side effect, but not really. I mean, it really does synthesize the partitioning of an array around a particular number the way you have to in quick sort. All of the lists in scheme, except for a very small subset of operations that I do not teach you, so you pretend that they're not in the language. All of the lists are immutable. That kind of coincides and makes [inaudible] without side effect very easy because if you can't change the list and you can't change the atoms it's very difficult to program by side effect because you can never change a list in place. You always have to synthesize a new one out of old ones.

Those of you who have programmed in Java before, their strings are considered to be immutable. Like you can't actually go into an existing Java string and change an E to an A. You have to generate a new string out of it. Does that make sense to people? C++ and C you know very well how you can change memory behind the back of another variable. Java makes that very difficult, at least with strings, not with objects in general, but with strings. Scheme makes it difficult in general based on the subset that I've taught you. Okay? Make sense? Yeah?

Student: Did you saw it was or wasn't a built in?

Instructor (Jerry Cain):When I wrote it in the handout I thought it was not a built in, it is a built in. I think it's called remove. I forget what it's called because I wrote it from scratch so it's remove in my head because I just use that code.

Student: How does it work if there's [inaudible]?

Instructor (Jerry Cain):Yeah. I actually was specifically saying I'm only dealing with – the remove that's built in does remove all the duplicates. The one I wrote. And I think this is why I ended up – actually, now I'm remembering I think this is why I wrote it. It is my version just removes the first element. The first version of it, but I'm actually prescribing as part of the problem that I don't have duplicates in the incoming list. Okay? Okay.

So let me give you a little bit. We all kind of gravitate. I know you were completely humiliated by assignments one and two in terms of their difficulty and how much they were exposed raw memory, but now I think we'll all kind of gravitate toward how things work behind the scenes and under the hood. I could give you a very simple set of drawings to give you an idea as to what these lists look like in memory. I did a little bit last Friday, I think, but I'll give you some more. When you type in – I'm being very informal here. I'm not actually drawing out the pictures that are really relevant to coa. I'm just drawing out pictures that are accurate enough so that they're a complete enough design, so that you can know that these things can work.

When you type in a four the scheme interpreter is prepared to build a variable in memory that stores that four. What basically happens is it recognizes as it parses it that it's a pure integer. It has a four up top. It actually returns this right here. So it returns a pointer to the data structure that is self-typed and self-identified as an integer and it levies some type of print operation against this thing as part of the read, evaluate, print result loop. Does that make sense to people? Okay.

So the reason this prints out a four is because the thing that is returned by evaluation is expressed as this pointer and so it goes to this thing and says, oh, it's a four, so that's what it's gonna print out. I'm gonna print it out according to what this data type is. When you type in hello it returns a pointer to something that's tagged as text or a string or whatever and how it does this – hello, rather. Rather the actual details are up to the interpreter, but as long as they return this to the read, develop, print loop and print knows how to deal with pointers to these types of cells that are self-type identifying it knows how to interpret the rest of this entire thing. Does that make sense? Okay. Do you have a question right there?

Student: Up there is some kind of encoded the type of data so first strings or schemes we have some kind of –

Instructor (Jerry Cain):All of the data types in scheme are dynamically typed. Java does the same thing actually. The data actually carries piggyback and not information about its own data type. I don't know. You haven't dealt with this part of Java so much. I don't know whether you're into Java or not specifically. I'm assuming that you know a little bit. There's actually a method that you can invoke against all Java objects called get Class. And it returns a class object. So all of a sudden it's getting very meadow on you because you're like, okay, class is sort of things we code up as templates and we actually construct objects in the image of these class definitions.

But programmatically you can actually have a class that models the notion of a general class. Okay? That's what Java does. Java actually has this data structure and every single object has a hook back to the class class that describes the class that the algorithm is a type of. Okay? Scheme isn't quite that sophisticated as far as I know. It may be now, but what I'm talking about right now, and I don't think coa is any more sophisticated than this, is that it actually tags every single data element with information about what the thing really should be interpreted as. Okay? So this obviously prints out hello.

When you type in – and this is the part I'll leave you with right now. I'll write some more code examples on Friday, but I only have like five minutes to talk about this, so I'm just gonna give you some pictures to mull over. When you type this in not surprisingly it returns this. That means that the thing is returned is part of the read, develop, print loop has to be the address of the leading nodule list and the list has to know it's a list, so it knows how to print itself out.

This is constructed on your behalf. I'm just drawing a linked list. I'm gonna draw this a little bit differently. It's not required to do it this way, but this is just the way I like to do it. This is associated with a one, this is associated with a two, and this is associated with a three. Okay? All I did there was draw a linked list. Thing is that's interesting is that when scheme digests this right here it knows how to programmatically – like, it's almost, like, it's reading a data file where you've happened to express your sequence of numbers using scheme like syntax and that behind the scenes every time it hits an open paren it knows it's gonna be building a list for you behind the scenes.

This type of drawing is consistent with the way I drew this four up here. These things right here, they're intended to be the node to the linked list. They're actually called cons cells. Okay? And it's not a coincidence that we have a cons function as well. This right here is understood to be the car field. This is understood to be the cdr field. Okay? When you levy a car against this list right here it actually constructs this thing right here and the car operation is just instruction to go in and return that right there. Okay? And then print it out. It's looking at a one that knows it's an end so print it as a one. When you get that right there, because you've levied a cdr against the list 1, 2, 3, it gets back the list to three. It's self-typed as a list, so it knows how to print itself out and it involves parentheses and it involves the two and the three. Okay? Does that make sense?

Stand-alone just returns this. Each of those cons cells is actually tagged with like the cons word, just like texting into there. Okay? Make sense? What this really is is this. It's functionally equivalent to the following. When I type this in it's really just as if you did this. Cons one onto the front of cons two. What you get by consing three. You prefer this for obvious reasons. But this can just be taken as a syntactic sugar for this right here. Okay? You can think about this being framed in terms of lots of elements and the base list, which is right here. Okay?

Cons as a scheme symbol is attached with code. It's native to the interpreter that knows how to basically mallic or operator new these thing right here. Does that make sense? And after it does that it has to figure out what to put there and what to put there. Oh, it just puts that in the car field and that in the cdr field. Does that make sense? Okay. And this entire thing evaluates the same thing that that does. Both of these, no matter how I type it in, comes back with 1, 2, 3. Okay?

So obviously there's a lot of implementation detail that's being left out, but if I basically charged you with the task as a final project, so just go and write a very miniature scheme interpreter. Okay? Then you could do it based on what you know. You could write it in Java or C++ or C if you want to use the vector. I wouldn't recommend it, but you could. You have all the rudimentary understanding of how the memory model of backing these things actually works. You don't know how function call and function evaluation works. How the code that's attached to a symbol is used to instruct how to crawl over all the remaining arguments and synthesize an answer. That's difficult. That's why every time I think about the beginning of the quarter I think about actually giving a final assignment where you write a scheme interpreter in scheme or you write a scheme interpreter in Python or something like that and I always revisit the function evaluation part and I'm like, ah, that's too much work.

But it's very easy to understand that it's possible to do it. Everything is framed in terms of a list behind the scenes. Okay? I have more to talk about memory management next time. In particular, I want to talk about how things are freed and how garbage collection, much like Java garbage collection, except it uses a slightly different model, how garbage collection works and I also want to talk about the equivalent of a dot, dot, dot from C and C++. I'll write some more code for that and we'll implement the generic map car of the generic map function. Okay? Have a good night.

[End of Audio]

Duration: 53 minutes

ProgrammingParadigms-Lecture23

Instructor (Jerry Cain):Hey, everyone, welcome. I don't have any handouts for you today. A couple of clarifications. Apparently, I goofed and I made the assignment due on some nonexistent day next week? It is due Wednesday evening, which is the 28th, not the 29th. That's ample amount of time to actually get the assignment done so I'm not worried that you're all going to be all of a sudden flustered, but it was always intended to be due Wednesday night. I think I even said Wednesday night in class; I'm sure I did.

But otherwise, that is due Wednesday evening. You will have one more assignment going out on Wednesday. It'll be just a single problem. And the language we'll probably start next Wednesday, if not in the last ten minutes of today, next Wednesday, where I start talking about Python.

Once I get to the Scheme segment, we're going to transition and lighten a little bit and we're not going to be so concerned about trying to master all these new features of language, because I don't want to say we've covered all of paradigms, that's not true. But we've covered a good number of paradigms, and when I teach Python next, I'm going to really just highlight individual things in the language that will get you up and running very quickly and point out where it's imperative and where it's object-oriented and where it's functional, okay?

So that's what we'll do throughout the rest of the quarter. I'll probably introduce you to a few of the languages that I happen to know, just so you can see them and really just trust that just because you don't know them that it's actually pretty easy to learn them.

Today I want to focus on Scheme. I probably will not keep you the entire time. I have two things I want to do and then I will probably just let you go and go on this virtual five-day weekend. Okay?

When I left you last time I was spending a lot of energy talking about the memory model, and I just want to review that for five minutes so that you understand exactly how you could very easily implement a Scheme interpreter.

When you type this in, this is really an instruction to just go ahead and synthesize a link list behind the scenes and equate it with the address of the leading note.

So I drew it very carefully last time, but this is going to be returned in response to typing that in and hitting enter, and then some print functionality is levied against this address behind the scenes, and it arrives at something loosely drawn, like this right here. And I drew it as if it were a nil object.

And I did this a little bit differently last time, but conceptually, that's what's there. And the print function can actually figure out, either because it's overloaded by strong type or it could actually look at this thing and determine that it's a list node, it can use that to figure out whether or not it's printing a standalone number or a standalone string, or an open param before it actually prints the leading list and things like that, okay?

When you do something like this, I've downplayed this. Let's just call it SEQ. Then very much of the same thing happens. Something like this would be constructed on behalf of that statement right there, and because the defined statement is one of the few things we've learned – in fact, it's really the only thing we've learned that has a side effect, it would associate in some global symbol table this thing called SEQ, and it would be associated and just put as an entry in some kind of global map.

So that any time you reference SEQ and they note that it's not a local variable, it looks in this global map to figure out what data is associated with it, okay? Does that sit well with everybody? Okay.

So let me just make it clear how lists are shared, and then because we don't allow lists to be updated, the aliasing problem is never dangerous like it is in C. If I go ahead and I ask for the car of SEQ, well, it evaluates to this thing right there, and the car is just basically a request to return whatever that is and it levies a print statement against that arrow or the base of that arrow, it detects that it's an integer type, it has nothing to do with lists, so it just prints a standalone one.

When I do this [inaudible] of sequence, it's just a request to go and grab that, okay, which itself is a list just like the leading list is; it just happens to have fewer nodes involved. So that's why it prints out two, three, okay?

The cons instructions are more interesting. If I do something like this, cons, and I'll do this – one, two, three, and I'll cons it onto the front of the list four, five, six, I will be exhaustive in the drawing here, what happens is this is synthesized, points to a nil – that's how I draw it, with a four and a five and a six, okay?

This is also synthesized. Points to another nil, okay? And then has this as a structure, and the leading arrows of these are just the product of the evaluation and the construction of those two lists right there, and then the con statement is actually just a request to build a new one of these to put the address of the first thing there, whether it's a one or a list.

Put the address of the second thing right there. And so when the address of this is produced by the evaluation of the entire statement, it does have the side effect of constructing this node right here, okay? It prints out this for pretty obvious reasons. It prints out the car, which happens to be a list, so that's why it prints that way. And then it prints out the [inaudible] in the form of a list like that, okay? Does that make sense to people? Okay, question?

Student: How do you know that the [inaudible] points to a list and not to, like, like –

Instructor (Jerry Cain):Well, the [inaudible] is supposed to always, at least in our subset of Scheme, it always points to a list. Now that actually isn't true in real Scheme.

You can actually construct what are called dotted pairs. There's no reason to do this. I'm only telling you because you asked. But if you wanted to do something like this, that would break the list paradigm that I've been pushing for the last three days. Turns out that that is legal, okay?

This would produce what's called a dotted pair, and the dot is in there to emphasize that the five isn't part of a list, that it is a standalone element. It's actually an integer that's in the [inaudible] field, okay? Without this, it means that the [inaudible] has a list, which points to five.

I'm not saying that this isn't important; it's just that I'm only talking about Scheme for five lectures, not 12. Some of the data structures that do exist in Scheme that I'm not talking about, they have the notion of associative maps. They actually store everything, all of their pairs behind the scenes, as dotted pairs.

And if this is incidentally, the values are lists, then the actual serialization and printing of it would look like a normal list. But if you mapped integers to integers, then you'd have all of these types of things being stored by the map, okay? Does that make sense?

Scheme actually has vectors, which are really the equivalent of arrays, and it actually also has these things called maps, okay, which [inaudible] a little bit of hatching behind the scenes, okay? Make sense? Okay, awesome.

So let me just emphasize a few things. I want to construct this list two different ways. Okay, I'm going to do one the obvious way. Right here, cons, one, two, onto the front of one, two, and this is an example, it's actually pretty much coincident with whatever's happening up there. But I want to draw the memory diagram out just to emphasize how the memory diagram changes when I write this a second way.

I would have this point to this right here, which points to – mm, oops. Yeah, that's right. Would point to a one, would point to a two, and this would point to an independent list with a one and a two. Sorry, I didn't mean to smush that, but it just kind of happened.

Okay? But you get that there actually are four-linked list – I'm sorry, there's five linked list nodes combined, okay?

You understand that. This right here, this is trickier. Here's the one, here's the two, and there's the nil. If I serialize and print out what's accessible from the tail of this arrow, I certainly print that out right there, okay? The printout rhythm is so blind to memory management that it doesn't actually care that there's some internal aliasing going on, okay?

The instruction for printing this out as the car and putting that out as the [inaudible], it's exactly the same. And the fact that they happen to reach the same nodes in memory is kind of just incidental. It would still publish this way. Does that make sense?

Now the reason I get this and this as separate is because I actually put down the two list constants, okay, like that. If I wanted to actually construct this, I'd have to be a little bit more clever. I'm actually going to go with the more clever of the two ways I'm thinking about it. What you could do is make sure you only construct one list from a list constant. That's going to create just this right here.

And then do this – lambda of X cons X on X, and make sure the argument is a list so it can be the second argument of the cons that's inside that lambda, okay? So I construct one list and I actually bind it to one variable that's used in two different places inside the definition, okay?

So this really is an instruction. Don't think about it from a data standalone. Understand that X and X are both associated with references to structures, and it's saying to populate the car and the [inaudible] field of some new con cell with exactly the same tail, okay? That's how you could do this, okay? Make sense? Okay, good.

So what I want to do, before I talk about garbage collection, which will be kind of high level and just a lot of drawing – it's not really difficult, but it's actually pretty interesting to understand, I wanna go over what somebody reference in a question on Monday, the equivalent of the dot, dot, dot, okay?

There's not much reason to learn the dot, dot, dot in a course that teaches Scheme over a course of five lectures, except for the fact that it is a paradigm class and we're trying to compare and contrast features of languages. It also allows me to implement the full map operation, okay?

We've been pretending, from an implementation standalone, that map is always unary map, and I wrote that last time, I'll write it again in a second. I actually wanna write generic map. That takes some function object and either one or two or three or four lists, depending on how many arguments the actual function object can deal with.

So we're pretending – we've been pretending that something like this – car, one, two, three, four, five, six, seven, was the only type of way you could use map. Using it as a unary function – car is certainly a unary function – and this would produce one, three – oops – five, and all the [inaudible] would be ignored. But again, just consistent with the idea of a map, it transforms one list of length three into another list of length three.

If I do this, I want it to just deal. That's right. I type that in, I want it to generate 111 and 422, okay? To emphasize the fact that this doesn’t have to choose whether or not it's being used as a three-argument function or a four-argument function, I could do this. I could map times over the list one and the list two and the list three and the list four and the list five.

I'm not saying that this is the way you'd want to realize that [inaudible] but nonetheless, this is supposed to trivially collect all of the cars of the list, seven of the peers of arguments, apply this to that set of arguments, and come back with, as a list, the number 20 – 25, where am I getting that – 120, okay? Does that make sense?

I have a ton of examples on the very last page of I think handout 32 about how map can be used as the generic map, okay? Not a ton, but just enough to illustrate the idea.

When we wrote unary map, I think on Monday, maybe early Wednesday, I really only handled this case, okay? I called it unary map specifically to emphasize that this was covered but these were not, okay?

I want to do two things. I want to reproduce – define, I'm going to call it unary map to remind you that it's unary – function, and then I'll call it list. No, I don't wanna call it that, I lied. Call it sequence. And if it's the case that I have the null sequence, go ahead and return null. Otherwise cons onto the front of some recursive result, whatever you get by applying fn, the car of the sequence, like so, to the recursive result. Unary map of the same function, to the list excluding the [inaudible]. [Inaudible] unary cons if [inaudible]. Okay?

Now you can see from the calls up there that I have to accommodate either one or three or five lists. I am going to force map to deal with at least one list, because it's really kind of silly to support the notion of a map over no list whatsoever, okay? But in order to accommodate a variable number of arguments, you have to do the equivalent of dot, dot, dot, but in Scheme.

As an aside right here, let me just talk about what happens if you define – I'll just call it some dumb name like bar – and I'll give it these arguments – A, B, C. That means that any call to the bar function I'm defining right here has to take three arguments exactly at the moment, unless I do this. And that's fine.

The dot in this context just means that everything that comes after argument three, like arguments four and five and six, if there are that many, that they should all be collected and put into this thing as a list, called D, okay? Does that make sense to people?

So if I do this, if I just define bar to be this, to be the listification of A, B, C, and D, list just basically takes all of these elements and wraps an extra bookend of parentheses around it, okay? So if I call the bar function on one, two, three, four, five, six, the output of this – it's a little weird, but it listifies these four elements – one, two, three, and D is then equal to the list that contains these three elements.

That's how one additional argument can accommodate everything in the dot, dot, dot sense. Does that make sense? Okay.

So the product of this thing right here is [inaudible] would be that right there. Okay, now that's not all that sexy of an example, but it motivates the dot and the way of catching an optional number of arguments for the implementation of generic map, which I'm going to do in a second.

I should emphasize that if you do this – bar one, two, three, you're actually gonna get one, two, three, empty list, okay? And if you try to pass fewer than three arguments to bar, it's gonna choke for the same reasons it would choke if there weren't a dot there, okay? You have to have at least as many arguments as are required.

So the implementation of map I actually think is very, very cool but it's very, very dense and it uses apply and it uses mapping, it uses unary map, and it uses function objects and things like that. It's a great function, and this is going to be the prototype for it.

The reason I wrote unary map is because I actually use the implementation of unary map in the implementation of generic map. Define – I'm just gonna call it map, we'll pretend it's not a built-in. We've just discovered the idea of mapping and Scheme and we're implementing it for everybody.

You can take map, you have to take something that evaluates to a lambda or function object. I'll take something I'll call first list, which is required, and then I will actually have a final argument called other lists, which because of the use of the dot – that's supposed to be all one line – because of the use of the dot, a second and third and fourth and fifth list will themselves be bundled in an additional list. So they actually are carried around by one argument called other lists, okay? Does that make sense?

I'm going to assume for simplicity that all the lists are of equal length. It turns out they real implementation doesn't need that, it just uses the smallest of all the lengths to determine how long the products would be, and if there are any extra arguments in some lists, they just ignore them.

But I'm just going to – for simplicity, I'm just going to assume that the length of first list is the same as the length of any list inside or all lists inside other lists, okay?

If it is the case that regardless of what the function is – null, first list – that I'm just going to assume that there's no mapping to be done. That for whatever reason I was given, maybe a ternary function or a function that takes ten arguments, but I was handed five or ten empty lists, or three or ten empty lists, okay? In which case the product should just be the nothing list, okay? That make sense?

Otherwise, what has to happen is I have to cons on the result, I have to figure out how to collect – in this case, this number was produced out of those three numbers. That make sense? Okay.

Obviously, I'm going to plus these. These two lists, in the context of this implementation so far, first list would be bound to that, right? And other lists would be bound to this, okay? So there's a little bit of asymmetry in how the data's handled, but that's just what you have to do, okay?

What I wanna do is I want to take the car of this, I wanna cons it onto the front of whatever I get by unary mapping the car function over this thing right here – clever that it does that, okay? Does that make sense to people, though?

I needed somehow to get the ten and the 100 out as peers in order, in a standalone list, so that one can be smushed onto the front of it so that I can apply the fn function, whatever it is, to the list that's been collected for me, okay?

So I wanna cons whatever I get by applying – I'm gonna need – I'm gonna run out of space. Apply fn to the consing of the car of first list onto whatever I get by calling the unary map this – whoops – right there is a helper function right here that knows that the list it's getting is – it's just getting one list. That ends the unary map, that ends the cons, that ends the apply, that ends the cons. Oops, sorry, I don't wanna do that. Oh, not that one, yeah. Sorry. Okay.

So do you have faith as to why this is gonna work? You may ask why I need the apply right there, why don't I just invoke fn like the function it really is? The problem is that fn, look at the plus. Plus doesn't take a list of integers, it takes a sequence of integers that follows it with no intervening parentheses, okay? Does that make sense? Okay.

So the apply has to basically get the plus or the car or the cons or the multiply onto the front of the list that's assembled by this thing right here. That make sense? Okay.

Then what has to happen is I have to cons this onto the front of what I get by recursively mapping this function over all of the lists where they've all been replaced by their [inaudible]. Does that make sense to people?

So I have to somehow get – let me bring – did I erase it? [Inaudible] Even though I'm handed the data this way right here, I have to somehow assemble the data for the recursive call to look like that and that and that, okay? I have to make them all peers again for the recursive call to map to work out. Make sense? Okay.

So the second argument to this cons, I'm actually going to have to indent over here, so just really assume this right here is just exploding over here so I have space to write everything, okay?

I certainly have to unary map [inaudible] over other lists, okay? That at least gives me the list with the 20 singleton and the 400 singleton inside, okay? What I can do then is I can cons the [inaudible] of first list onto the front of that. That at least gets all the [inaudible] to be peers, okay? Does that make sense to people?

So there's that. Then I also want to cons fn onto the front of that. So what I'm basically doing is I'm setting up – I have this asymmetry with the lists, I made the argument, the [inaudible], symmetric, and then I have to get fn onto the front of it, because fn isn't actually what's being invoked right here, it's map that's being invoked.

So what I can do here is I can apply this map routine that I'm writing to this right here, okay? Does that make sense? So in the context, let me just – I can actually trace to this, I think it'll be pretty good if I do it. Let's concern ourselves with just enough of a trace to convince yourself that this is gonna work, when I do map – and I'll just do – I'll map the list function over one, ten, 100, two, 20, 200, and that – I should do one more – three, 30, 300.

This is supposed to give me three lists back, the list one, two, three, the list ten, 20, 30, and the list 100, 200, 300, okay? Make sense? Okay.

So the initial call actually has to take this one and the – I'm sorry, the initial call actually takes the first piece of data and binds it to the thing called first list. Everything else is bundled in a second list. So those are my two arguments to this function, okay? What this does right here is it takes that and conses it onto the front of this, so that unary map call actually produces a list two, three out of this.

This one is the car. The first innermost cons call actually gets the car onto the front, okay? And then when I apply fn, I get the fn on the front, which is listed in this case. And that's how I get the list one, two, three out of it, okay? Does that make sense? That's the easier of the two to follow.

What has to happen for the recursive call to map is that this has to have [inaudible] mapped over it, like that, okay? That's what the unary map does, the second unary map thing does, okay? I have to cons the [inaudible] of this onto the front, okay, and then I have to actually put the function object, whatever it is – in this case it's a list – onto the front of that so that all of the arguments that are peers at this level are once again peers down here, okay?

And then I can use apply of map, so that the map is effectively put onto the front and it's invoked as if we typed it in that way, okay? This is dense, but I think you're all getting it, okay, because I see nods. Okay, so you're all with me?

Okay, that is more or less the implementation of the real map function. The only thing that I'm not handling is what I alluded to earlier is the variable length lists, but that's not really interesting. That just means that there's more base cases right here, and this is more involved. It has to check whether or not the first list is null or any one of the lists in the others lists is the null list, okay?

So it's just another unary map, basically, to figure out whether or not there are any falses or any things in there that pass the null question mark test, okay? Question?

Student:[Inaudible]

Instructor (Jerry Cain):Oh, a left bracket. Which board?

Student:[Inaudible]

Instructor (Jerry Cain):This one. What did I – a list bracket – you mean just more parentheses right here?

Student: In the middle. After the first [inaudible].

Instructor (Jerry Cain):No, no, no, no. These right here, this list is consed onto the front. So this used to be the list, the product of the last of all the cons calls right there. Does that make sense? And then I consed whatever fn was in this example of list onto the front, okay? And the four arguments that were passed to the initial call, one, two, three, four, are sort of peers again in this list right here, okay?

They've all lost – well, these three have lost their cars. This doesn't – it's not supposed to lose anything because it's supposed to be passed through verbatim, okay? And then when I apply the map, this effectively takes this onto the front and the map says okay, there's the function object and there are three peer lists after it, okay? And it invokes it like the original call was made.

Student:[Inaudible]

Instructor (Jerry Cain):No, no, no, no, no. This right here, because I consed the [inaudible] of the first list onto the front of it, it made – took the asymmetry where this was in a list of lists and this was just a standalone list, and it put this list onto the front of this one, so they're all peers. So there shouldn't be any extra parentheses in front of the 20, okay?

There was, but then I pulled that parentheses in like it's on a spring and I smushed in the list ten, 100, okay? Does that make sense? That's kind of what I think about cons doing.

Okay, you all happy? Now, the one thing you're hypersensitive to from assignment one through assignment six but not at all sensitive to in assignment seven is dotted memory allocation and deallocation. The only places where you allocate memory in Scheme, it's not on your behalf. It's when you type in list constants and it builds lists behind the scenes.

Or technically, you can say that you're actually dynamically allocating memory when you use the cons function, because you know that that actually creates a cons cell and populates its two fields with two pieces of data, okay? Does that make sense to everybody?

What about deallocation, right? That's actually more involved. Well, very often when you just type in expressions – when you do something like this, when you do let's say cons, hello, onto the front of the list there, you don't actually store the result of that list anywhere in memory. You just for whatever reason had decided that this is how you want to construct the list hello there just long enough for it to be printed so that you can see them as peer words in order so that it reads "hello there," okay?

Behind the scenes, this is constructed on your behalf. The hello and the there are a little bit more involved, but conceptually you can just assume that what they address are Scheme strings. You're returned this right here, okay? And then after it uses the tail of that arrow to figure out how to print the list hello, space, there, it actually, in principle, could deallocate the list for you right then and there, okay?

It can recognize from the statement as simple as this one that there are no side effects whatsoever. Like the result of this list isn't being attached to some variable via define. It's not being passed to another function. It's just this little hiccup of a statement right there that actually prints something, and then after that happens, the memory can be reclaimed. Okay, we're all in agreement?

Well, actually, reclaiming it, that's algorithmically difficult when you learn about length lists, but eventually you get very good at it and you know exactly how to do this breadth-first reversal of the list and free all the nodes along the way, okay? I mean, if you wanna think about it, you could do this – you could just say to free anything is to just free that, free that recursively, and then free the cell. It's as simple as that, okay?

It's hard for you to do that in 106b when you first learn it, but eventually you get very good at these things and you just kind of do it, and once it's done once, you just forget about it because it really basically covers the deallocation of everything.

There are situations, even in the Scheme that we're learning, where the length lists that are built up actually do have to persist for a little while, but it also wants to be able to detect later on whether or not a list has been orphaned so that it can actually free it once it does detect that it's been orphaned.

So let me just forget about functions for a minute. They're their own beast. Let me just use define and just define some global variables, which comes up more often than Scheme – no, I don't wanna say more often, but certainly comes up and we use a few globals in where am I, but not very many.

We use globals but we define constants, which is what you're supposed to do when you're dealing with globals. But just imagine the scenario – and I'm only interested not in the elegance of the coding solution but how memory management works – if I do this, then you know that there's a link list that's formed in memory and the tail of the leading arrow is bound to and associated with the symbol called X, and it's stored, permanently or semi-permanently, in a map.

And I say semi-permanently because obviously the map comes down when you quit [inaudible]. And it also can change if I were to redefine X somewhere later on, okay?

But on behalf of this, X is associated with some list one, two, there, period, okay? Do you understand how the node that holds the one and the node that holds the two and the node that holds the three, that they all can't be reclaimed right now? Because I can refer to X anywhere down the line until [inaudible] quits, okay?

Let's say I do this: define Y to be the [inaudible] of X. All that does, because of what [inaudible] X evaluates to, is it basically points to that. I know it's obvious to you pictorially, but I'm going to depend on this observation is that one is addressed by just one thing, okay? The two and the three are effectively reachable from two different symbols, okay?

So I go on and I include a few statements that have no side effects that are of interest to the drawing, and I go ahead and say you know what? I don't like the fact that X is defined that way. I want to associate it with the list four, five. So X stops pointing there, and it points to this list four, five. I can even see the one, it's like over, like, the edge of a cliff there, the way that the drawing works out, okay?

So if whatever Scheme environment you're dealing with is detecting that memory is dear and it actually would like to clean up orphan memory for you, deallocate the memory, do what's called garbage collection – that's the terminology that's used in all modern languages, really.

It really should understand or have enough of a structure in place to not touch the four and the five, and not to touch the two and the three, but to reclaim the one if it wanted to, okay? Does that make sense to people?

So just in this drawing, 80 percent of the con cells are still actively – are still reachable from some symbol, okay? One of them, 20 percent, is not reachable, okay?

The system could just leave it orphaned for a while, but eventually, when maybe it's waiting for IO or it's waiting for a network connection or doing something where it actually doesn't have anything else to do, it might say okay, well, since I'm not doing anything meaningful, I'll go and do garbage collection.

So you might ask well, how could it actually do that? Well, one technique is that some systems might actually go ahead and decorate these nodes with what are called reference counts. Those aren't little commas and dots, they're actually zeros and ones, okay? Before X was reassigned up there and it was pointing to this, I would have had one and two and two there. Does that make sense, okay?

I'm sorry, I would have had one and two and one, because this would be two because the one and Y would be pointing to it. But this would still be one because only the three is pointing to it. Only the two is pointing to it, rather, okay?

Student: The problem with reference counting is that the real version of Scheme actually allows you to update the cells of list nodes or cons – I'm sorry, update the current [inaudible] fields of con cells in place. So you actually can program with side effect, and that basically means that if you want to you can create a circularly linked list, okay?

And that would mean everything in that list would have a mutual reference count of at least one, even though it might be this orphaned ring of friends, okay? Does that make sense? So reference counting doesn’t actually work if it wants to be able to reclaim everything. So what happens is that this is very high level, but I think conceptually, this is the better takeaway point, and it's not important that you know how to implement it down to the semicolon.

Every single time you call cons, either directly or it's called on your behalf because you’ve just typed in a list constant or you even use a define – define a factorial as a function – it actually stores the code in list form.

But you can imagine there being something of a master con set, okay, where every single con cell that has been created but yet to be deallocated is just catalogued somehow behind the scenes, okay? And maybe this actually points to a one, and maybe this points to a two. I'm sorry, maybe this points to a two and that points to a three, and this doesn’t point to anything.

Oh, I'm sorry – this points to two, because it's a one, but nothing's pointed to the one. So I'm being a little bit more elaborate in my drawings of that over there, does that make sense?

This could be a four, this could be a five. This could point to nil, I should do this the same way. When I talk about, from a data structure standalone, this master con set, it actually is something like a hash table or a binary search tree, so that it can actually find any single one of those in a breadth first reversal or a depth-first traversal or just a clear mapping over that set data structure.

When we talk about this symbol table, or this symbol map, where things like X are associated with that right there, okay? And things like Y are associated with that right there, okay?

So I'm just being a little bit more structured in my drawing there of the two lists that are relevant to that rightmost board over there, okay?

Now how would this as a system know that this con cell can be reclaimed? The fact that it's pointing to two is of no interest to anybody whatsoever except this orphaned one field, okay? And it's not like just because it has a hook on this two field, but anyone can discover it. It is really gone, okay?

But this and this and this and this all have to be preserved, okay, they cannot be freed by some garbage collection process that's running in some low priority thread in the background. Well, the algorithm that works beautifully in the Scheme setting, okay, and it actually could technically work in any setting if you can actually control how assignments are done and who actually gets – how pointers are handed out, what could happen is the low priority garbage collection thread could say okay, I'm about to go and clean up the garbage, okay?

But in order to do that, I'm gonna go through this three-stage process. I'm gonna go and I'm going to assume that every single one of these things can be freed. Okay? And it goes through and it marks every single node. Attached to some node is some bullion or some little bit of one or zero that's set to zero, meaning that if I ever see you again, or I see you in the near future again, and that bit is set to the founding face, then I'm going to assume that you can be freed because no one cares about you, okay? Does that make sense?

But then, before it actually goes and sweeps the garbage, it comes and says okay, symbol table, but here's your last chance to go and save whatever con cells you think are important. So from this standalone it says okay, I'm going to do a depth-first traversal of everything that's accessible from any node in the symbol table here, and this is gonna traverse and say you know what? No, you gotta keep that and you gotta keep that and you gotta keep the four and the five that are associated with it, okay?

Does the same thing for the two and the three, and in the process, because one is truly orphaned, it's not reachable from anything that's in X or Y, so the third of these three steps, master con set says okay, I gave you your chance I hope you were good about it and accurate about it, because I really am going to go and commit to deallocating – operator deleting or freeing, however – or letting garbage – Java's garbage collection do whatever it does to go and reclaim that right there, okay?

So the actual con cell set, that's not real terminology. I'm just making that up so I can call it something. It's only slightly more bloated than it ever needs to be in terms of overallocation, but it always has the option, if it feels that it's too bloated, to go through and clean exactly what isn't needed anymore and spare everything that is needed at that point. Does that make sense? Yup.

Student:[Inaudible] the master con set, instead of just marking everything that needs – that [inaudible]?

Instructor (Jerry Cain):Well, the problem is is that – actually, you technically don't need the first pass if every single time you create a new con cell it's automatically marked with the frowny face, right? I'm just worried about the situation where something like this might be created during the sweep, okay? So it's probably better to actually, when this thing is allocated, it's allocated for a reason, okay? And that the smiley face bit, or the don't sweep me bit, is set to be high, okay?

Because at least initially you're optimistic in saying this is probably going to contribute to something for a while, okay? In that case, if you just can't trust what these values were ahead of time, then the initial sweep has to go through and say I'm marking all of you for deletion, okay? But there are certainly ways of optimizing it.

I haven't thought all the way through because I haven't actually implemented something like this yet, okay? Does that make sense? Okay. Yeah, go ahead.

Student:[Inaudible] not necessarily work because [inaudible] all the false – first [inaudible] that you created them, and then you do a sweep then at that time that it's being marked by another, like by a variable? And then so you set it to say no, don't clean me up, but then it gets changed before the next sweep, so it would say oh, no, don't clean me up, even though it's still not being [inaudible]?

Instructor (Jerry Cain):Well, no [inaudible] happens in this binary [inaudible] critical region series, so that con cells would not actually be created during the sweep process. Now that's counter to the defense I was giving as to why by default I would associate this with a smiley face or whatever. That just might be a heuristic, to say I'm assuming it's in use because I'm actually defining it right here.

But I mean, basically you're about a [inaudible] condition threat, and it could just solve that using concurrency, like we learned a couple weeks ago, okay? You guys are good?

Okay, so [inaudible] curious, I know a lot of you are just doing nothing at all this weekend because you're going to be so bored, so if you want to learn about other languages that are purely functional languages, at least most of the purely functional, just go to Wikipedia and look up – I'm just throwing these at you; I'll actually talk about one or two of these in the last few days of the course – look up STEAM, look up ML, look up something called Haskell, which I'm still planning on bringing someone in to talk about.

And then there's this other language which is absolutely hilarious called Erlang, which actually I use – well, I don't use but a lot of people at work use because not only does it have really good functional characteristics to it, it also happens to deal with distributed systems very nicely. Kind of a weird mix of features, but that's why people use it.

Scheme you know a lot about. ML, short for Metal Lizard – don't ask. Actually, it's short for Meta Language, but I always say it's short for Metal Lizard, because that's the name of some interpreter where I learned ML. ML is a strongly typed version of Scheme. It has a different syntax, it uses square brackets instead of parentheses in a lot of places, it has a different notion of cons and car and [inaudible] in how it actually dissects lists and builds them up again.

But as opposed to Scheme, which is, like, I don't care about types, it's only if I'm actually trying to do something where I depend on the types that I'm actually going to holler with some kind of runtime error, ML actually has a little bit – even though it's purely functional, it has a little bit more of a compile time element, because when you define functions, as opposed to Scheme, it'll infer type characteristics about everything that's making up the function.

So if you do something like X plus two, it can see that two as a token is an integer, which actually constrains X to be an integer if it's going to be compatible with the plus function that deals with the integer domain. Does that make sense?

And then it can detect if the function itself is one that takes one integer and maps it to another integer, and that is itself a data type. It's kind of like a function pointer type that knows how to take two void stars and return it into another void star, or something like that.

So Scheme is functional, weakly typed; ML is a functional language, and it's strongly typed. I don't know very much about either of these two languages. Some of my TAs do. I just know that they are more the rage recently. I don't know whether it's because it's fun or they think they're great languages, but these are other buzz languages that come up in programming language circles these days.

I don't know any – I'm sure that there are practical uses of Haskell or else it wouldn’t exist. I don't know what the motivating factor for designing it was, I just know that it is a functional language. It's very Pythonesque in its functional language characteristics. We'll learn about Python come Wednesday.

And Erlang I actually want to just do some research and tell you what that is about if we have the time. But nonetheless you have this little bullet list of things you can look up over the weekend if you're curious about what other functional languages there are out there, okay?

So there you have that. There is not going to be any discussion section on Tuesday, okay? I mean, we're done with Scheme, you’ve actually seen the set of very difficult problems in Tuesday's discussion section, and you have the assignment due on Wednesday night. So I'm not gonna have section on Tuesday because there's really nothing to cover. I haven't talked about Python yet. We will have a discussion section during dead week, okay?

So have a good weekend I will talk to you later.

[End of Audio]

Duration: 50 minutes

ProgrammingParadigms-Lecture24

Instructor (Jerry Cain):Everyone, welcome. I have two handouts for you today. You all know that Assignment 7 is due this evening, 11:59. It is the last assignment that we'll issue a letter grade on. Like Assignment 1, Assignment 8 is intended to be this kind of like transition type of thing. We're trying this out of the course. I'm not planning on putting Python or anything related to Python on the final exam. I'm just trying to fill out the course with a new interesting language that's topical. It certainly represents a paradigm, but I'm just trying to get in the habit or not forcing lesson material on the last few weeks or the last few lectures to come up with a clean endpoint, which is probably I'll call last Friday as far as testable material, but still fill out the quarter with an interesting assignment. The assignment that's due a week from today – I'm sorry, a week from tomorrow. I'm giving you until Thursday evening of next week. It's this very short Python assignment that's just intended to get you used to the framework. It's not gonna teach you everything about the language. It's really just taking a program that works and making it work a little bit differently just so you're forced to deal with Python and its integers and its Booleans and its tests and to fully define functions and classes and things like that, okay.

So as far as how the rest of the quarter will work out, I will definitely lecture today and Friday. I probably will lecture Monday because I don't think I'm gonna get through all the Python material. As to whether or not I lecture next Wednesday, I think will just depend on how quickly I get through the Python material that I want to introduce to you, but I'm not trying to race as much material as possible into the course because I know you all have a lot of other things going on. Fortunately, 107 does not have a lot of stuff going on in the next week. After you get the scheme assignment done, then you're really just in a situation where you can start studying for the final, okay. I'll have a practice mid-term – I'm sorry, a practice final probably out next Monday, a practice solution as well. The mid-term is scheduled for June 9. You can take it either at 8:30 or 3:30 on June 9. I have rooms set up. I forget where they are. I'll put that on the practice exams. Okay, so this thing called Python. Oops, let's spell it right. There are two overarching features that I want to emphasize. It is as opposed to anything you've seen before, Scheme you could argue is a little bit like this, but C and C++ and Java are certainly not. It's an example of what's called a scripting language.

Okay, and then as far as that, we can call that a paradigm. It is also imperative. It is also object oriented. It is also functional or it has functional components in the language just like Scheme does, and I'll show you an example hopefully today, either today or the beginning of Friday where you actually see the equivalent of map, okay, in Python, and you're all intimately familiar with the notion of map right now. You've seen vector map and you've seen iterators and things like that in C++ and you certainly know map from Scheme. There really are lambdas. In Python, there's also the map function. There is the notion of an evaluate statement just like there is in Scheme. So I put these up here kind of to remind you that imperative is the C language, it's the assignment oriented, function oriented type of language, I'm sorry, procedure-oriented type of language. [Inaudible], you're very familiar with that. Everything's centered around the data and we call objects for classes and functional is an example of a paradigm which Scheme adheres to. I would say that Python is object oriented like C++ is. A lot of people program in C++, not so much as C++ programmers, but as C programmers, but like to use objects. Does that make sense?

And they still had this very sequential way of thinking and they organize data in terms of classes as opposed to structs. I actually – I get the impression based on what little Python programming I've done and what good amount of Python code I've seen is that it really is the case that people program imperatively, okay. They rarely subscribe to the functional or object of paradigms. They usually go with this imperative approach where they have a series of tasks that need to be done one after the other, and they just get them done in bullet point format, okay, and subscribe more to the C like or Pascal or just the imperative approach of actually accumulating data piecemeal until you actually have a result at the end. They may incidentally use objects, certainly those that are part of library set that come with Python. They may like to use lambdas and they see an opportunity to use some functional paradigm approach to solve some subset of the problem. They might do that. I really think of it as this where it just happens to have these features inside of it. That's consistent with the fact that it's a scripting language.

The best analogy that I can think of outside of programming for scripting analogy is if you actually ever go see a play and for whatever reason there's like a casting emergency and somebody who normally plays the part and has the lines memorized isn't available, so somebody has to actually fill in the part and they have to hold the script in their hand, okay. I have seen that a couple of times. They have to go through the motions. They have to do everything perfectly, but they actually are reading their instruction sets while they do it, okay. Do you understand what I mean when I say that? Okay. Just think about a set and like a dress rehearsal for a play where you still have a script in hand. I think of a Python script or any kind of shell script for that matter as being very much like that. It's this very what you see is what you get. It digests function definitions and it digests individual statements as they read them and as they read them the side effects of their execution are actually realized while the script is running, okay.

So it's like Scheme in that regard. We normally prepare all our functions and put them in a dot SCM file ahead of time and then load them and then run the result. That still has a scripting feel to it, but the emphasis is on the functional paradigm. With Python, and a lot of things, Pearl, a lot of things that are designed to be run a script, to batch process these together to get some large set of tasks done. I actually see that – I see Python doing that more than anything else. I don't see many games written in Python. I don't see many compilers written in Python, although I do see some servers written in Python. A couple of – I don't want to say the names because I'm not actually sure of them, but there is this set of companies right now that are setting up services, not these companies specifically, but companies like Twitter and Friend Feed that are all trying to basically broadcast information. Some of them, I'm not sure which ones. I think of one, but I don't want to say it and be wrong, are actually writing all of their server and services in Python, okay. Normally those things have been written in Java or in C++. Python has a very rich set of networking libraries and natural support for SMTP and ACTP and ACTPS and all of these networking protocols that are very common today with all of these distributed systems that are set up to provide these types of services, and I know Python is contributing to that.

So there really is some large-scale software that's being written in Python beyond the scripting set that I'm gonna focus on today, okay. As far as other buzzwords that describe Python, it is dynamic. Scheme is also dynamic. C and C++ are not at all, okay. They have a huge compile time element and there's very little type information at run time. Scheme and Python both maintain type information about every single piece of data that exists in the run time, and you can actually query the data type if you want to, okay. Does that make sense? Okay. It is really interesting to code in Python because it feels very much like C and C++ and Java in the way that you just do block structures and if statement and Y loops and things like that. But there's a down play on the emphasis of curly braces and parenthesis, okay. There's a – the block structure is actually decided, not by curly braces or by parenthesis, but by white space and tabs.

And it's the most frustrating thing to get used to the first day you program in Python because you just naturally put in curly braces, and then you realize you do that, and you're like, "It's not gonna work." So when I show you our first – show you the first Python function I want to write, you'll see just that it's just about taxed with very little delimiters. There's double quotes and things like that, but there's no curly braces. There's very few parenthesis. It's really just very strange to look at the very first time you see it, but then you just get used to it after you program in it for a while. So what I thought I'd do is I'd take the Scheme approach to introducing Python and just show you the Python environment, how you can actually talk to Python and get it to announce – like tell you what the sum of the first four integers are and what happens when you type in a string constant and how Booleans work and things like that, just to you get used to the syntax, and so you have the understanding as to what things look like. And I will do that first. Going to the screen, I will try and get as much of this to fit on the screen as possible. Okay, that's not bad at all. Okay, so I launched Python much like we would launch Kawaa, okay. I just type in the word Python. As opposed to Kawaa, Python basically comes on any system that has any Unix backing whatsoever. So the mess, the pods, all of them have Python installed already, so does Mac OS10.

So you just go into a terminal and you type Python and it's probably gonna do what it just did right here. This part should feel very familiar to you. I type in a four and it comes back with a four. I type in hello there, and it comes back with that. It happens to delimit it in single quotes. You can actually use them interchangeable. It allows you to use both double quotes and single quotes because you might want to delimit the entire thing in single quotes if there are a lot of double quotes in your string constant and vice versa. That doesn't mean that a lot of modern languages are paying a lot of effort to because there's so many quotes that appear in HTML documents and things like that they don't want you to have to backslash and escape everything because that just makes it that much error point of an investment to try and code something like that up. It actually has [inaudible] numbers not surprisingly, and that it is also nice that we're back to normal addition, okay. I don't want to say that this is conceptually confusing, but whatever.

We're used to end fixed notations, so Python went with the in face approach. As far as the Boolean constants are concerned, true is true, false is false. True and true is true. You actually spell out the keywords, okay, kind of like you do in Scheme, but they're in face again. If you want to – I'm trying to think of what else.

The one thing about strings that are interesting is that there really is no direct support for the notion of a character. It's pretty clear that this is a string the way I type this in, whoops, quote A, B, C, D, E, like that, okay. This happens to be the best you can do for isolating an individual character. You just think of characters not so much as characters as you think of them as strings of length 1, okay. Does that make sense? Okay. Numbers themselves are just plain old numbers, but strings, let me do this. Hello starts with HB. Whoops, didn't like that. Oh, that's my fault. Hold on a second. Oh, I'm sorry. I'm just doing something. I'm sorry, starts, doesn't like that, sorry. I'm just messing up here. There it is, okay. Capitalized in Scheme, okay. The one thing that's unique, and this isn't unique to Python. I've seen this in C sharp and managed C++ as well is that you don't necessarily have to create an object to surround an object constant in order to send messages to it. Does that make sense?

I certainly could have done this. Greeting is equal to hello, and then done something like this, greeting dot starts with worth, that right there. And that's the way you would have taken – would have done it in C++ or Scheme or something like that, but Python is just smart enough to know that if you're sending a message to something and it's actually a constant, it can tell by looking at the constant what data type it is, so it can just build the synonymous object around it so that it really can receive these messages. Does that make sense to people? Okay, so as far as all the primitive types, I've more or less touched on everything I wanted to. With regard to Booleans, you have and, or, and not spelled out. That shouldn't surprise you. True and false are the actual Boolean constants. I will do this. You may look at that and you may say, "Okay, that doesn't make sense." But do you understand that as opposed to pure Scheme where everything evaluates to something, that thing right there didn't evaluate to anything. It had the side effect of actually associating the number five with X, but X equals five didn't print anything out. Does that make sense? That's because it evaluates to this constant called none, which is basically the equivalent of void from C and C++, which means there's really no side effect or nothing of interest in the evaluation of it, okay. Does that make sense?

So you can still do this, okay, but then it adopts a whole different set of functionality in order to propagate the assignment. But the overall assignment and actually evaluates to this constant called none. And none really doesn't mean very much except that it doesn't have anything printable about it, okay. So there's that the – as far as strings are concerned, there are a whole set of methods on the string class. It's certainly a superset of the set of string methods you see in traditional languages like C++ and Java. Python recognizing that when it was written, it was intended to be something of a web savvy language. There are some very like very strange methods that exist for the string class.

Let me see if I can do some of them. Hello, there. Capitalize right there. That's a strange method, but it just realizes that a lot of things get printed sentence wise as part of HTML documents, so rather than actually having you go through and uppercase everything that comes at the front of a string, it just has built in code that does this, okay. It has some really weird stuff. Let me see if I can do this. Is title, it comes back with false. So a really weird predicate method to be built into the language, okay. It really just analyzes a string and it goes through the entire thing and sees whether or not every single word that's delimited by white space or by the boundaries of the entire string happens to begin with a capital letter. So something like this as if it's a movie title – like clearly Iron Man is a title. And I'll go back, don't worry. Just like that comes back with a true because it's clearly the name of the movie, like it just knows, okay. Does that make sense to people?

Now I'm not clear what the motivation is to why these things were specifically included in there except for the fact that they just wanted to have – maybe it must have been the case that they just saw these types of things being written several times in other languages and said, "You know what? Let's just take care of it. Why make them rewrite it again? We'll just put it into the core language, okay." Does that make sense? Now as far as writing functions is concerned, it's a little difficult to do at the command prompt for the same reasons it's difficult to write a Scheme function at the command prompt. I do want to introduce one other data structure before I write a function, okay. Let me show you what a list looks like. This is a list constant just like that, okay. The square brackets actually mean something. It means not only is it a list, but it's gonna be a mutable list. So I can do something like this, small nums is equal to one, two, three, four, five, six, just like that. And then I can ask what small nums of zero is, okay. And that doesn't surprise you. I can also do that this is the fun part. What do you think that means?

You're right because it actually understands that it is just – there's no reason to buy a search from the front in terms of the syntax that's available to you, so it recognizes that you might be interested in the last character, but you don't want to do it like this where you do lin of small nums minus one. I'm not even sure this works. Yeah, it does, okay. Lin I've used on strings. I haven't used on list before. You could do it that way, but then it actually has to manually count the entire thing, whereas internally they could just optimize operator square bracket, the C++ terminology to just have access to both sides, and when there's a negative number to just start searching from the back instead. Okay, this is – I mean the neat part. What I can do is I can do that right there and I can get sublists. Does that make sense to people? Okay, so that wants everything from index zero up through, but not including index three. If I do this, that's basically the equivalent of that right there. If for whatever reason this programmatically comes up, I can get the empty sequence, everything from index zero up through but not including index zero is just basically the empty list.

If I'm interested in the last – the end of the list, I can just punt on the thing that follows the colon. It basically says negative two up through the end of the string. Does that make sense to people? If I want everything but the last two characters, I can do something like this, okay. Does that make sense? So it's the syntactic sugar that was introduced to the language because this language is much younger than any of the other languages we've studied. Scheme is really the oldest of all the languages we've looked at, at least in this class. Scheme, we're talking like late 1940s, I'm sorry, late 1950s, early 1960s when there was a lot of research circulated around it. It was really introduced to kind of emulate the lambda calculus and come up with an implementation of that, and it was used for symbolic programming and larger programming and things like that.

C was invented for the purpose of implementing Unix. It's like they wanted access to the hardware. They wanted some form of abstraction. They thought it was brilliant level of abstractions at the time, I'm sure. We know better now, but it actually provided a much cleaner syntax than what was otherwise available to those implementing operating systems, which was assembly code, okay. You had to get down there in the nitty gritty of the hardware. It's hard to do that in Fortran or Scheme, so they actually – they. Those who worked on Unix invented the C language with the intent of actually making it easier to build the OS. C++ was invented late '70s, early '80s. It became fashionable around 1990. It was initially used as C with objects, and then a lot of effort was put into the implementation to introduce templates and a string class and all of those things that you're now familiar with. But we're talking with a language that at this point is 27, 28 years old in terms of specification, okay.

Python is probably 13, 15 years old, okay. It didn't come into any level of popularity until year 2000, 2001 is when I started to hear about it. That doesn't mean that it's all of the sudden popular only because I've heard about it, but when I was doing a lot more consulting work at the turn of the century during dot com era, I heard of a lot of big name companies starting to use Python. Google is the one specifically that I remember hearing about so much that people that went there to work said, "You know what? You should really teach Python in 107 instead of Scheme." I'm like, "No, I like Scheme better. I think it's cooler as far as illustrating a paradigm." But I'm getting to the point where Python is something we have to pay attention to because it has that much more of a presence in industry, okay. If you want this, just another neat little feature of list, you can do this with strings as well. Hello, there. See if this works. Oops, sorry about that, totally can. Okay, it emphasizes the fact that it really is a sequence of characters, something of a list. It turns out as an immutable list of characters. Strings are immutable like they are in Java. The one thing that's really fun, let me just do this. List – I shouldn't do that. Let's do SCQ is equal to – let's do zero, one, two, three, oops, can't count. Do that right there, okay, and I want to go back and correct the problem, okay.

Do you understand that the part where there should be a four is actually the sublist of SCQ and I want to say it's this right here. I may be messing this up, but I don't think I am. I've just identified the splice within the list that's missing the four. Does that make sense? Okay. So what this is gonna do is it basically comes up with an L value and identifies the sequence in the list that is synonymous with four, up to, but not including four, which is this empty region between the three and the five, and you can actually assign to it. Does that make sense? I hope I'm not messing this up, but we'll see in a second. There you have that, okay. Now it turns out this is just cool. It's not anything you can't do in Java or C++ or Scheme already. It's just that it's more gracefully handled by the syntax that you're seeing right here, okay. That's just the product of it being a more modern language. It can learn from all the other language's mistakes and know where a program was cumbersome before and just try to come up with some better solution now, okay.

There is a variation on the list. When I do this, SCQ is equal to five gate – whoops, commas are important now. When I do that, I actually have the option of taking sequence and saying, "You know what? I have fives. I'm gonna replace it with a 14." And so we're dealing with a clear script of what's called a mutable list. I just call it that. There is a variation of the list which was immutable. I do this. It's also considered to be a list, but the parenthesis, for whatever reason, I don't know why they've dumped the parenthesis, but that marks – I mean is considered to be an immutable list, a collection of data that is read-only. So if I do this, that's all fine and dandy, but if I try to do this, it freaks, okay. It's actually not called a list. It's called a tuple. I'm reading that right now. And but it's clearly marked behind the scenes as read-only. It just wanted to be able to make the distinction between something that can be updated while the program is running and something that is supposed to be just this bundling of data in some predefined sequence that can't be updated at run time. So it's just basically this one little gesture toward const or final in the language, okay. Does that make sense? Okay.

So as far as lists are concerned, they are in fact objects. You haven't seen any object oriented flavor to this yet, but if I do SCQ is equal to – let's just do some strings, which is what these really are. SCQ append D works, okay. Does that make sense to people? Okay. So there's that – there's a whole list of list methods. There's actually lists, there's sets, there's dictionaries. They're all these things they'll talk about in a few minutes. But let me just go ahead and just write some code that does something algorithmic. It's not gonna be very sophisticated, but I just want to show you what the things look like. I do have to kill the computer for a second. I will come back and show you how it works afterwards. As far as languages go, this language is more like the make file than anything I've ever seen before, and it'll be clear in a second when I actually write some code. I want to write a – I think a fairly juvenile function in terms of algorithm, but I want to illustrate the syntax. I just want to write a function that's in the handouts called gathered divisors.

So I want it to be able to work like this. I want to be able to pass, let's say, gather divisors, and I want to be able to pass in something like 24. And I want it to be able to spit back 1, 2, 3, 4, 6, 8, 12, and 24 is fine with me, okay. I want it to accumulate all the things that evenly divide into the number that's specified, I'm sorry, all the positive integers that divide evenly into it. Keyword is def, okay. Just know INE was too much work to include that, so we just go with the EF, gathered divisors. I'll just call it number, and there's a colon right now, which means it's very clear about that. Everything that follows, any line that follows this one that is indented is under the jurisdiction of the entire definition, okay. Does that make sense to people? So what I want to do is I want you to just assume that the number is a positive number. There's some error checking in the handout. I'm not gonna worry about that right now. What I'm gonna do is I'm going to set divisors equal to this list constant. There are no semi-colons. I almost put a semi-colon there. It's very hard not to put a semi-colon, okay, because you put a semi-colon in every other imperative language other than Scheme. It just knows that the lines end and now backslash N means semi-colon, okay.

More so than any other language this space right there, this tab right there, okay, is absolutely required because if it started right there, it would have not only marked a statement that said divisors is equal to the empty list. It would have implicitly marked the end of anything associated with this definition right here, okay. So all the white space that you see in the handout in the definition of this function, every single one of those, it's not four spaces, it's actually a tab. It may be stores as four spaces by the editor, but it's rehydrated into a tab or the equivalent of a tab when you actually load the file again, okay. Not surprisingly, what I want to do is I want to look over every single thing between 1 and 24, okay, and inclusive 24. And I want to decide whether or not the number defies it evenly. I'm speaking like you're all 106A programmers. I know you know how to do this part. You just may not know the syntax within Python to do it.

This is the way you do it. I'll say dif in range. I will say zero. I will say number plus one plus one. It looks very for loopish. It certainly is. It has a technically different approach to the way that it generates for loops. You're used to four I is equal zero, I less than ten, I++, and there's a test that is checked with every single iteration to see whether or not you should continue. For loops are almost always expressed – I'm sorry, they are always expressed in terms of what are called iterables. That means that the object of the N key right here has to be something that evaluates to a list of things, okay. Now this range function is a built-in. I'll explain – I'll type it on the console in a second. But this actually evaluates to the list. I don't mean zero here. I mean one. It evaluates by default to every single number between and including this one up to, but not including that one right there, which is why I have the plus one there. So this actually evaluates to the list, one, two, three, four, all the way up through number, which in the context of this thing would be 24, and we want 24 in there, okay, because I want it to appear in the result.

There's an optional third argument right here that can be the step amount. By default, it's one, which means just include every number in counting up sequence. If I put a ten there, it just would have like done 1 and 11 and 21, etcetera. You can even make it negative if you want to, if you want to generate a list where it's counting down from some high number to a low number, okay. This idiom right here, it's technically different than the for loop you're used to. It just iterates and it associates div with every single number in this iterable, okay. So in the first iteration it's associated with a one and a two and a three, but for different reasons, okay. If there's some implicit – there's no implicit plus equals one or plus plus coding behind the scenes. It's incidental that the numbers are sequential because that's the way we constructed the iterable. Does that make sense? Okay, so there's that.

So with each iteration, this is the easy part. If it's the case, I need a tab there. I don't need parenthesis. It doesn't cause problems, but you don't need them. If it's the case that number mod div equals equals zero, then I want to take divisors and I want to append whatever div is on this particular iteration, okay. And I have no semi-colons. When I come right here I'm gonna just return whatever's accumulated over the course of that iteration divisors. There is no ambiguity as to where the if test ends. I'm sorry. There's no ambiguity as to what's under the jurisdiction of that if test, and there's no ambiguity as to what's under the jurisdiction of that for loop, okay.

The fact that this came all the way back over here means that it's a peer in the sequence of statements right here and this marks the end, not only of this if jurisdiction, but also of the for jurisdiction, okay. If I had moved this over one tab, it would not have been under the jurisdiction of the F, but it would have been under the jurisdiction of the for, which means that it would have returned after the first iteration was over, okay. Make sense? Okay, so there's that. I'm going to store this in a file called divisors dot PY. The dot PY is more required than you'd think. It's actually not required, but I think every single Python file I've ever seen ends in dot PY, okay. So what I will do, so I'll bring the computer back. I'm doing okay on time. This is good. And I will control the exit because I want to see where I am. That's not where I am right now, CD devel, CS107, Python examples, okay. Let me just – I have a better idea. This more divisors right there, you see the gathered divisors right there, so there's clearly some codes in this file that looks very much like the code I just put on the board. Let me open it in a slightly prettier environment.

Text Mate is the coolest little program ever. Does anybody use Text Mate? It's great. Okay, so here's this. Let's just focus on what is visible, nice soft autumn colors, okay. I actually did not include the 24 in my implementation here, but I included it on the one on the board, but conceptually it's exactly the same. The weirdest thing about this is you see this triple double quote at the front. If the very first statement in a function definition like I have here is a string, it's understood to be a documentation string. I'll show you how you can actually find that in the run time interpreter that I was showing before. The triple quote means that the string I'm about to present is actually expected to appear over multiple lines, so don't give me a problem because I don't put a double quote at the end, okay. Does that make sense? Okay, so there's that.

As far as this is concerned, that's the entire function. I have some other code in here. I didn't put this in the handout, but there is prime. There's scattered primes, whatever. It's the same exact idea. So that's the entire thing. There's no real analog to this in C. There's a little bit of an analog to it in C++ where we have name spaces. You're much more familiar with this from Java where you import packages, okay. Do you understand what I'm talking about? Okay, there's a little bit of that in Python as well. The fact that I called this gathered divisors, of course that's relevant if I want to actually call this function. The fact that it's inside this thing called divisors dot PY means that it's inside a module called divisors, okay. So that when I come back and I quit this and I go to a directory like that and I relaunch Python.

Actually, I'm sorry. I didn't mean to do that. CD devel, it says 107 Python examples. Okay, I'm inside this directory that has all these modules of code. I'm gonna go ahead and invoke Python. And if I try to call gathered divisors 24, it's like, "What are you asking me? I've never heard of gathered divisors. It's not in the language. Yeah, it may be in any one of the 15 trillion files in the world, but you have to tell me which one it's in." There's a couple of ways to do it. Import divisors does that. That is relative to the current execution path of the current path where you – I'm sorry, the current directory backing the Python run time. It basically goes in an digests everything inside the Python file, so it's the equivalent of the load statement from Scheme, okay. And you say, "Okay, that's great. I can just do gathered divisors, and now it will work." And the answer is no, it will not. The reason is just because you've digested all the material that was inside the divisors module, yeah, the divisors module, doesn't mean that all of the sudden all of the things that were defined in there are the master copies of code attached to all those symbols. So if you really want to invoke the gather divisors, there's two ways to do it. The more clumsy way is this. Divisors dot gather divisors, oops. Did I actually get that right? Yes, 24. Pray with me, yes.

Okay, so basically the module is the name space, okay, and you have to frame by default all functions you invoke in terms of that name face so it knows specifically which gathered divisors you're talking about. You may think that's ludicrous, but in our real system, you're gonna have presumable, let's say, between 1 and let's say 4,000 Python files that are all contributing to a system. You have to make sure that no two functions are named the same thing because if they are you want – but if there are, you have to qualify which one you're referring to by actually including the packet name inside. Now if there's no danger of ambiguity, you can do this. Let me control D and start over. You can say from divisors, import, gather divisors, and then all of a sudden gather divisors is a top-level function name. So it's just syntax. It's all in the handout. It's not really the emphasis, but I'm just leading you through the full example and hitting on everything that I think is important, okay. Does that make sense to people? Okay, now you know how in Scheme that you modeled everything in terms of lists, and if you had a struct, you'd say, "Oh, we'll just use lists." And the CAR of the list is the name and the CAR of the CDUR is the GPA. You just had to remember the actual structure that was imposed in your lists. Python you don't need to do that.

You can elect to do that if you want to, but most people use classes, which I'll talk about a little bit more on Friday how they work. They're not complicated. It's just that we really do have better modeling schemes available to us in Python because it's just a more modern language and they included those things inside of it. What I thought I would do here is I would pretend that we haven't done that stuff, and I will introduce you to, I think an even more central data structure to Python than anything else involving lists or sequence or tuples for that matter, okay. There is a – the notion of what is called a dictionary. I'm very careful to use the word dictionary in Python because that is the replacement word for map. You can use map and if we're talking about Python, people would know what you're talking about, but dictionary is the operative word for the primary data structure in Python. If I do this, let's say I'm student, and right now it's not a very interesting student. I use square brackets and parenthesis for lists. I use curly braces to delineate all of the content of a dictionary constant. And when I do that, all it means is that the student is an idea at the moment, okay.

I can do this though. Student name is equal to let's say Linda, okay, and there's that. And now all of the sudden student has grown quite a bit, okay. Make sense? I can do this.

GPA is equal to 3.98. Go, Linda. And then we have that, okay. Does that make sense? If I want to say I forgot her GPA I can do this and I'm reminded, okay. Interesting to know is that the dictionary is completely backed by a hash table, okay. And it's a very small hash table, and when I say small, I meant the number of buckets is actually very, very small initially because most of these dictionaries imitate structs and classes and objects in Python. In fact, objects are really dictionaries with just a little bit more embedded inside of them. If I want to update for GPA because it's just not high enough, that's Harvard. That right there obviously updates the dictionary in place, okay.

The reason I'm talking like this is because the dictionary is easily the most malleable, easily manipulated data structure in Python as opposed to lists in Scheme where you pass around lists and you functionally manipulate them to create new lists. Python, at least to the extent that I'm gonna have you exercise it in Assignment 8, really just deals with strings, which you already have enough information about, and dictionaries, okay. There's one thing I forgot to mention last autumn when I taught Python for the very first time, and it impacted people on Assignment 8, so I just want to say it right now before y'all disappear for he day is that all of these objects, and I say objects loosely. Anything, any aggregate data structure like a list or a tuple or a dictionary, they're all passed around by reference. So if you write a function that takes a dictionary, it just makes a shallow copy of it. Any changes that you make within the function are reflected in the original. Does that make sense to people?

That's a huge point for the purposes of getting through Assignment 8 because you're gonna use one of these dictionaries to keep track of a master set of information that accumulates through some intense recursive algorithm, okay. And it just involves some caching of previously computer information so you don't make the same recursive calls several times, okay. So that's why I wanted to say that right there. I think it's pretty clear that the dictionary right here is heterogeneous in the fact that it really doesn't impose any requirements at all on what the values need to be, okay. So I have a string attached to the field called name. I have a double or floating point attached to the thing called GPA. If I wanted to do this, share, and there's that. Whoops, what did I do wrong? Oh, I'm sorry. I meant an array. There's clearly order, student, and then all of a sudden you have this third thing inside here. So you can attach anything you want to these things. The fact that it was printed first just means it happened to – that friends happen to hash to a lower value. That's really what it is, okay.

If I want to bundle all of my ideas, but I don't have any at the moment, I can do that. So there's this one thing where one of the keys happens to be associated with an interdictionary, okay. Does that make sense to people? Okay, so it is really free form. In fact, let me just do something like this, playground is equal to initially it's empty. I don't recommend this. I'll show you the problem with this in a second. It actually doesn't have a problem, okay. So you can have things that are non-strings. It just – anything that is hashable can actually appear as the key inside a dictionary. I don't recommend actually mixing them. I would stick with strings, okay, for the keys, but it's a freefall as to what value you want to attach to those things, okay. Does that make sense? Now come Friday I want to do a lot of things. I'm going to jump ahead to the lecture next week. I forgot that it's – I thought it was gonna cover like this entire handout today and I'm just not – I just didn't at all.

I want to talk about a lot of things about Python specifically that are interesting from a language standpoint. The libraries are certainly interesting, more so than even Java, which you have some familiarity with. It has incredible support for XML processing, for HTTP processing, for building a web server, things like that, just amazing that you can really – of course it's rudimentary, but you can build a web server in about 15 lines of code, and I'm not overstating it. And it's not very sophisticated. It just does the special files and it doesn't do anything clever about like compiling pages or building pages dynamically, but if you just want to fetch pages behind the scene, you can use Python and about 10 lines of code, 15 lines of code, do exactly that, okay. That's usually a statement that the libraries are really taking care of something and they're taking care of it very, very well, okay. On Friday I want to show you a little bit more about dictionaries. I have this very dense example, but nonetheless, I think representative of how Python would solve the random sentence generator problem that you all solved for Assignment 1, okay. The difference is that you actually prepare the grammar and you frame it in terms of a dictionary, okay.

You may think that it's cheating that you're getting the data structure in memory, so you don't actually have to worry about reading it in from a file and parsing it and looking for the angle brackets and things like that. We didn't have the option in C to do anything else, okay. We don't have data structure constants that you can actually very easily assign to variables in C. You have to build them up out of raw deserialized data, okay. In Scheme and also in Python, functions and data – I'm sorry. Functions like the ability to express a constant carried beyond just the set of integers and the set of strings. You can define list constants. You can't do that in C or C++, okay. You can define tuple constants and you can define dictionary constants. So you can just elect to format the RSG grammars in something that kind of looks like a Python dictionary so that when you set it equal to a variable called grammar, it's loaded, okay. Does that make sense? And then you can just write the recursive code and frame it in terms of this variable that you know happens to be a Python dictionary. So you can use operative square bracket and you can assign things to it and you can make random selections from the list of options that are available and associated with each non-terminal, okay.

And I'll go over that. It's in the handout, but I think it's very nice to look at. It also introduces map, this Scheme functional language thing as to how that can work. Okay, so with just a little bit of work you are more than outfitted to tackle this Assignment 8. Even if you struggle on the first like 30 minutes to an hour just to get the syntax down, I think you will recognize that it's not intended to be a lot of work. It's just about getting used to the Python idea, living without a compiler or an interpreter that does very much checking for you. Okay, I will see you all on Friday.

[End of Audio]

Duration: 48 minutes

ProgrammingParadigms-Lecture25

Instructor (Jerry Cain):Hey, we're on. Look at that. How'd that happen? I have one handout for you today. Actually, I don't expect to get to the material today, but I don't see the disadvantage in giving it to you just in case you're bored over the weekend and have nothing else to do, you can read about XML and Python and networking.

What I want to do today is I want to focus a little bit more on dictionaries and show you an example where dictionaries actually contribute to a meaningful program. I'm gonna re-write RSG from a sign of one in Python. We're gonna do it in laughably little space. I'm gonna be able to illustrate a very small program that has the imperative, object oriented, and the functional paradigm all in it. Okay.

I'll illustrate lamdas that they actually exist in Python and it's really just a matter of coming up – knowing the different syntax, the Python way of doing what you did in Scheme a week and a half ago or just until I guess last night or two nights ago rather. I also want to talk a little about objects and classes in Python, and talk about the object model.

It's very interesting to look at it because when you learn about objects and object orientation, and in a class like 107 where you look under the hood to figure out how these things are implemented, you can very easily get the impression like I did for years to be honest, that all objects, and everything is implemented the exact same way in all languages, and that's just not the case.

So I will talk a little about that today. With regard to dictionaries, I mentioned on, I guess it was Monday – what's today, Wednesday, sorry, Wednesday – that dictionaries are a central data structure in Python. They are basically a very simple syntax layer over what is very well understood to be a hash table.

They keys can be anything that are hashable. They don't even have to be heterogeneous. You can have some integers and some strings as your keys, and the things that they map to, the values, they can be any type whatsoever, and they don't have to be consistently the same type over the dictionary.

It is very amorphous, very heterogeneous, just like Scheme is with its data structures. You can put anything on a list, you can mix up data types you can do the same thing with Python structures. Okay.

What I want to do is I want to emphasize the fact that as opposed to C and C++ where you don't have a clear way to texturally initialize a data structure – like think about a C++ map or a C vector from Assignment 3 looks like after you've populated it with say 20 values. It actually depends on the binary representation of the data. Okay. And there's no way to say C++ vector of int is equal to 12345. Okay.

You don't have object literals in C or C++ or for that matter Java either. Okay. You can have array literals in Java in a way that you can't in C and C++, but the real sophisticated containers, there's no real way to actually specify container constants. That's not the case with Python.

The first of three or four pieces of the RSG example – I won't write it out in full grammar, but what I'm basically doing here is I'm coming up with a file format for a random sentence generator grammar. You are familiar with this. I know it's been eight weeks ago, but you remember that it involved some angle brackets for non-terminals and things that were terminals were just normal strings. If I do this grammar I'm actually writing Python code.

Now I will make it clearly a dictionary. I'm going to have grammar that is just a gesture to what the full grammar could be like, but my grammar is going to be expressed as a dictionary literal where the keys are structured this way, and they map to arrays which are called lists in Python, or at least the way I use them I call them lists, lists of expansions. And for a start, I'm only gonna have one expansion.

So here's the list. It happens to be a list of a length of one, and that item is itself a list, and I'm gonna do it this way. It's not a very sexy sentence it's just a placeholder. This object is here. If I had a second or third option, they would have been common delimited list. I'm not gonna have that for this one. I'm just gonna have one option just so you see the structure in a small example, but I will let object map to a couple of options.

It can map to the standalone thing like computer. I'll let it map to this par, this assignment. I'm not interested in grammatically meaningful sentences; I'm just interested in getting the grammar on the board. I think this is complicated enough. You look at this and you may think that there's really no other sensible way to do this, but realize that kind of like you have in Scheme that you are allowed to represent the data in serialized object form. Okay.

And that this actually gets grammar to be an in-memory dictionary where it has two keys and each one maps to a list of lists where this one happens to be a list of length one and this happens to be a list of length three. Does that make sense? Okay, I'm being fastidious about the white spacing just because I'm careful about it in the handout as well.

What I want to do is I basically want to conceptually expand the start symbol to generate a random sentence, and knowing that it's gonna select this, I'm gonna want to expand this as a terminal, I'm gonna wanna expand what's in there, I'm gonna wanna expand what's in there. This and this are actually simple, it's just supposed to print. This is supposed to recursively do the same thing as if object is the start symbol, and it should expand to whatever it's supposed to expand to. Okay. Does that make sense?

Forget about the libraries, let me just invent function names. They happen to be the real function names. We're generating random numbers and names and things like that. Okay. If I assume that grammar is global variable. Just assume it's a global variable. We'll correct that in a little bit. Just because we're dealing with Python doesn't mean we should just be lazy about globals, but just to illustrate all three paradigms at once right now. I wanna define this function def expand and I'm just gonna call it symbol.

Now you know I do one of two different things depending on whether or not the symbol which I'm just gonna assume is being cast in as either a terminal or non-terminal, and if it's a non-terminal then it's definitely in my grammar. If it's the case that symbol starts with this string which is incidentally the best way to represent a character. If that's the case then what I want to do is recognize that I don't want to print to the non-terminal, I want to select one of its definitions from its definition set randomly, and then basically expand, in order, all of the terms that make up the list. Okay. Does that make sense to people?

So all I'm gonna do is this. Definitions is equal to the global grammar of this thing that I'm assuming really in the grammar and no semicolon. Okay. That brings in this or this entire thing. Now there's a built-in which I'm just gonna do this. I'll just say expansion is equal to – there's a built-in function called choice which takes a list – it actually takes either an integer or a list. I'm gonna give it a list. Choice is basically a get random function in Python as a built-in. If it's given an integer it gives you a number between zero and that integer exclusive just with random probability equal distribution.

If it gives you a list it selects anyone of the elements from the list with equal probability. That's kind of what I want. I want it to choose this, this or this with probability one third. I want it to choose that with probability one. That's exactly what this line is gonna do right here. Okay.

Now on Assignment 1 when you did it using C++, you did not necessarily use recursion. You probably used the iterator to just visit everything, but if you're thinking in terms of Scheme and how it dealt with lists, we didn't use iteration there. We didn't know about mapping technically when we did Assignment 1, but on behalf of something like this, if I understand that expanding this space will just print this space, and expanding space is here period we'll just print to that after I get the else clause up here. Okay.

And I can also recursively expand that. I can do this map, the expand function, over what's locally recognized as the expansion so far. Okay. Does that make sense? So if this is capable of being levied against start with an angle bracket, it's certainly capable of being levied against that, and I'm just gonna implement the else case where it isn't a non-terminal to just do sys out write.

That's basically the equivalent of print out for [inaudible] and less than without any new lines, it's a raw character printer, where I will just print out the symbol. Ultimately every single non-terminal becomes a terminal or a series of terminals to be printed, so this is what's going to be doing the printing. Okay. Does that make sense?

Object orientation, clear imperative style, and I am electing to go with this functional Scheme-ish approach of actually looking at everything that's making up the expansion list as peers and having them all publish themselves whether there's recursion involved or not. Okay. Does that make sense to people?

As far as how to do this from the get-go, as a global function, I could actually call the seed function that just basically is like randomized from the CS106 library set. We're just shaking up the dice a bit so that we really general pseudo random numbers, and don't start from the same number every time we run it. And then I might do something like this expand – now the handout version is a little bit cleaner about how it actually paginates the answers and things like that, but that's basically the gist of it. Okay. Does that make sense to people? Yes, no? Okay.

So what I want to do now is I want to bring the computer up. I'm gonna talk about classes, and how to define classes in a second, but there's gonna be – I wanna emphasize some things that confused people last autumn when they did this DNA assignment, and it was more or less because I just didn't mention these things specifically last autumn.

It was probably from me teaching Python for the first time last autumn when I just didn't know what the problems to expect were going to be. This is in the handout. There's a clear output. You can actually run this. This is up in the CS107 Website underneath WWW you can go and actually find this if you want to and run it. Okay. Any questions about this before I divorce myself from this example? Yep?

Student: So to do recursion in this language then you have to call map?

Instructor (Jerry Cain):Oh absolutely no. I just elected to use map. If I wanted to write Fibonacci I could have a definition of Fibonacci and call it Fibonacci. If I wanted to, I could have done a four loop from 4I in range of zero though length of expansion, and I could have just called expand inside a four loop. I don't have to use, I mean, we can always make a direct recursive call in virtually any language that I know of. Okay.

I just elected to go with the map approach here because I think it is cleaner, and it is kind of the functional way of doing things. Not necessarily that that's the goal, but I'm just illustrating all three paradigms in the same example. Okay. Question in the back?
Student:

Could you repeat what C does again.

Instructor (Jerry Cain):Oh, C right here. Computers even though they – when they generate random numbers, they do so deterministically. Okay. So they're not really random, but there as seemingly random as a computer that's completely deterministic in its operation can be. Normally what happens, every time you launch the Python interpreter it uses zero or something related to zero to generate the first random number, and then it algorithmically uses prime number theory to generate the next sequence in the random number sequence.

If you always start from zero then you get the first same random number every time you run something, and you get the same second number every time you run something. You get the same sequence, and it kind of breaks the fallacy – I'm sorry, it's exposes the fallacy that these aren't really random numbers. That's actually not a bad thing. A lot of times I recommend getting rid of that when you're testing so you do get the same output every single time, and so you can really debut a lot more easily.

But C just says, you know what, just populate – just set it so the first random number that's going to be generated is related not to zero, but usually this is in the case of C, and I'm sure it's the same in the case of Python that it usually takes the number of milliseconds or the number of seconds since the computer was turned on, and relates the first random number to that instead. This is basically the equivalent of s rand from C if you're familiar with that function. Okay. Yeah?

Student:[Inaudible]

Instructor (Jerry Cain):Yeah, they're right here. This seed and this choice, they're both from the same module, and technically if I'm complete about everything up top, I should do this. I should do import sys – that's short for system obviously. It's because of that that I'm allowed to go in and publish the console. And I could either import the random library and then do random.choice and random.seed or I could use the sexier way of doing it which is from random specifically import choice and seed.

And that's another way of doing it. Okay. It's just boilerplate, more boilerplate learning the language, things like that. Actually before I leave this example, I wanna do something. I wouldn't write it this way, I'm actually fine with the idea of grammar as a global here because it's just a little script, and it's really executing this as a statement, and then this as a statement, and then this as a statement. I don't really think of grammar as a global.

I think of the entire file as a function that it gets executed, but if you wanted to be a purist about this, and you didn't want this right here to be a global variable, you could pass in the grammar like that. There's a little bit of a breakdown because now all of the sudden expand is a binary argument function as opposed to a unary argument function. That doesn't mesh too well with the way I've called map. Map and Scheme actually can deal with multiple lists depending on the arity of the function that's being mapped.

That's not the case with Python. This has to be a unary function. It doesn't have to be a named function. Okay. If I wanted frame the implementation and I wanted to frame this function, in terms of expand, but as a unary function I can do that using the Scheme idea lamdas. Lamdas actually exist in Python as well. What I can do, it's just syntax, I can invent a function right here. I'm not just writing the Scheme code here, I'm really writing Python.

Lambda, I'll call it item for lack of a better word, and use a colon for the same reasons you use a colon every place else. Okay. And I just equate it with expand of item grammar. I'll abbreviate that. That's the first argument. Do you understand how that's a one-argument function? Just believe the syntax, it is right. Okay.

It is scripted as an anonymous one-argument function whose implementation is framed in terms of that one argument, and this thing that it available as a local variable in the outer scope, and I want to map that over expansion. That's another way to do this, and it really has the functional components of Scheme that are interesting to me, and probably to a lot of people as it is mapping, and it has lamdas and the closures that come with lamdas. Okay. Does that make sense? Okay, good. Okay.

So what I want to do is I want to talk a little bit about the object model from a memory standpoint. I'm gonna talk about this to a couple of degrees. I'm just gonna worry about dictionaries at first. The manner in which dictionaries are passed around is precisely, at least for our purposes, precisely the same as the manner in which objects are passed around in Java. Everything is always passed around by alias or by reference. Okay.

So if I do this, let me just deal with lists because they're easier to draw, and I'll just generalize the dictionaries. If I do this, it doesn't print anything out, but if I just print X, naturally it does this. If I do this, and then I print out Y, it prints out 1, 2, 3. Okay. That shouldn't surprise you at all. Okay. There's some languages where what I'm about to do would actually be different, but if I go ahead and call X, which is a list and I do append, and I append a 4, if I go ahead and print out X, you know for a fact that it's that, and that isn't the least bit surprising.

You've logically updated X. However, what you may not realize is that X was evaluated, and the result of the evaluation was assigned a Y. All X did was it evaluated to the pointer, to the lead note on the list or the lead element in array. So when I go ahead and print this out, I'm gonna get that because I changed Y behind its back when I actually updated X. Does that make sense to people? Yep, go ahead.

Student:[Inaudible]

Instructor (Jerry Cain):I'll get to that in a second. It doesn't happen by default. You have to invoke one of two functions depending on whether it was just a one-level copy or a full deep copy is available to you. This is one thing I did not mention last autumn, and it caused problems. My particular solution to Assignment 8 doesn't use any copying whatsoever. It just shared the same master dictionary throughout the entire implementation, but I should not have assumed that everyone would want to do it exactly the same way.

So you do need the ability, at least initially, until you convince otherwise, to be able to clone dictionaries. I'm gonna talk about that in a second. Let me give you some sophisticated examples. If I set Z equal to the list, let's say 10, 12, 14, and I go out and print Z, it's of course going to be this, but then I do this. I'll do W is equal to, and I'll construct a list that way. Okay. Z and Z evaluates to the list 10, 12, 14. Okay. So when I go ahead and I print out W, not surprisingly, I get this. Okay.

I bet everyone believes that. What you might not recognize is that it does not make any deep copies. You do not transfer full ownership of the Z lists into the list that's owned by W. So if I do this, z.append 17 and print out Z, it's now this, but more interestingly, if I print out W, I get this: 12, 14, 17, 10, 12, 14, 17. These are obviously contrived examples that are very, very small, but it's illustrating – for some reason to me at least it was more mysterious because it felt like it was this higher level language, you didn't necessarily know that objects were backing these things.

It was very easy to wonder whether or not it's a deep clone or a shallow clone, but almost always, at least initially, it just makes a shallow copy of it. Okay. It turns out that the shallow copy is preserved for the lifetime of the data in Python, and it happens to be different in a language called PHP which I use a lot at work where when you pass one dictionary around or one list around, it doesn't actually clone it, but it does actually label it [inaudible] it was copied from something else, so as you change it, it actually does a copy on write, and actually branches off the part that changes so that the two logically look different even though they're sharing memory. Okay.

None of that happens in Python though. Okay. If you want to make a copy, there's two ways you can do it. There is a module called copy. Okay. I suspect it's not written in Python. I'm suspecting it's written in C, but I could import something called copy. There's actually another function I'll talk about in a second. If you want to clone an object, X = 14, 15, 21, and you print out X, you will get 14, 15, 21.

If you do this, this is the same as all the prior examples and you print out Y, you get this, but that shouldn't surprise you. This shouldn't surprise you either. You don't know the syntax, but there's a key word in the language called is, X is Y is basically equality at the pointer level, and because X and Y really are aliasing the same exact physical memory, not only are they logic identical, but they're memory-wise identical as well, you expect this to come back with a true. Okay.

If you want to make a clone of something because you want to make some changes to a data structure without it affecting the original, you can do that. Z is equal to a functioned called copy, it's this copy right there that does it and I want to make a copy of X, not surprisingly Z is 14, 15, 21, but I'm out of room, but if I did Z is X, I would actually get a false back because they're memory independent at least to some degree.

Now copy, you would think because it's called copy, that copy really means decline, do this depth first reversal and make sure every piece of memory that's generated on behalf of Z is independent from that that was accessible from X, that's not the case. This particular copy is what's known as a shallow copy, it only goes one level down. Okay.

So if I had a list of atoms, like I do right here, these really are fully memory independent, but if I were to have lists of lists of lists of lists – does that make sense? The top-level list would be replicated from a memory standpoint, but everything inside would just be a shallow copy. So it's almost like it generates new memory for the top-level array, but that does a mem copy behind the scenes for everything below that. Does that make sense to people when I say that?

If you really do want a deep copy, then you have to use this strangely named function called deep copy, and this is an interesting example. If I do M is equal to the list 1, 2, 3, I do N is equal to the list M, M, and that's good enough actually. Recognize that I have the list 1, 2, 3 in memory. That's my abbreviated version of showing the link list in memory, and this thing is associated with the variable M. Okay.

The link list that's generated on behalf of N is really this. Does that make sense the way I drew that? Okay. It's the same list that appears in Index 0 and Index 1, but the same list is being pulled in two different scenarios. Okay. If I do this, P = deep copy of M, which is interesting to me, not only does it do a recursive descent clone of everything, but if there are any cycles like there kind of our in this, forget about M, that doesn't interest me anymore. Do you understand why those two arrows point to the same 1? Okay.

The deep copy not only figures out how to make a deep logical clone of the entire thing, but it actually figures out how to preserve the graph structure. Okay. Does that make sense to people when I say graph structure? Yes, no? Okay. So this would be associated with something that was completely independent, but would have the same aliasing internally. Okay. That's actually just a wise thing for it to do so that it can re-hydrate objects from their serializations if you ever wanted to go to that part of the Python language. Okay. Making sense?

Now it is – I'm thinking that you do not need to use these functions for Assignment 8, but whatever is said right here on behalf of lists, also applies to dictionaries. Okay. You use dictionaries and strings quite a bit in the Assignment 8 solution. You don't have to use lists all that much at all. You can if you want to.

I use dictionaries to more or less bundle information that would otherwise be aggregated in a struct, like C and C++ would require it, and I use that to kind of aggregate information that's related to one another. Okay. That's I think a common practice in a lot of these dynamic modern languages. Python, certainly Perl to some degree, I don't know Perl as well, but Python I certainly know that's the case. I also know it's the case with PHP. Okay. So there's that.

What I want to do now, is I want to show you a little bit about objects and classes in Python, okay, and show you how they really are very little more than just dictionaries. Okay. This is gonna be interesting because – it's gonna be interesting to me, and hopefully it's interesting to you as well. Remember when we learned about the object representation? I'm sorry, the activation record layout of a C struct, right.

You have a clear order in which fields are declared. You actually declare the fields ahead of time because there's a compile time element to it so you can bother doing that, and the first field goes at the bottom, the second field goes above that, the third field goes above that, etc. Okay. Objects and structs, classes and structs in C and C++, and actually classes in Java all adopt that model. All classes do is recognize that structs and classes can be the same thing, they both have data fields.

They don't store the method pointers or anything like that in the struct, they just lay things out according to the same exact formula. When you see something like that, you just assume that every language that's ever going to be invented from that point on is going to use exactly the same model. However, in a language like Scheme or Python, which have no compile time element whatsoever, you don't pre-declare the types ahead of time.

You just add stuff to these dictionaries which is basically Python's answer to the struct or the class. Okay. Does that make sense to people? And so you can't specify an order of fields ahead of time because you're not even specifying the fields ahead of time. You just kind of add things to the dictionary as it is suitable for your algorithm, and if you add X, and then Y, and Z, but in some other execution you add Y and then Z, and then X, it should still logically have the same set of keys, they just happen to not be inserted in the same order.

Python and most modern languages, and to some degree Objective C, that's the Cocoa language for Apple, uses this for storing methods. They actually just put all the fields, and store them as strings in a dictionary. Okay. That's exactly what happens backing for dictionaries, and also for classes in Python which are backed by dictionaries, and we'll get there in a second. Okay. So let me go ahead and just show you and example of a class, and I'll do some tinkering at the command line.

Okay, this is good enough, and let's just focus on this right here. I don't think I can – that's not good. Okay. I'll try to make the font bigger in the meantime. It might be too big, but we'll correct it in a second. That's not bad actually. Is it up there? Now it's huge. Actually that's okay. That's a nice word over there. Let's bring it and make it nice good word. Okay.

That's good enough I think. I'm not worried about the comment that's being clipped off, then I can bring this down, and that's all I'm interested in. Okay. That's great. Actually it's like artwork. Okay. You see the class keyword that should just be immediately obvious to you that it's going to be defining some class in whatever sense Python defines classes.

You see this underscore underscore, and init underscore thing, that just means it's a special method. Not surprisingly it's related to construction. I'll show you how to construct a lexicon object in a second. Remember how in C++, I probably said it this way about a millions times, and now it's a million and one, that it always silently passes the address of the receiving object as the negative one parameter, right.

Well, Python doesn't do that. It's very explicit about passing the address of the relevant container or object to all methods including the constructor. By convention it's called self, it doesn't have to be, but self is a keyword that's borrowed from Objective C, and I think that's right, yeah, actually it is. Objective C's self, it's basically just Python's equivalent of this right here. Okay.

Because objects are initialized dynamically, there's no compile time element whatsoever, it takes this empty object that really is a default object called object, that's part of the language, it's like java.lang.object, but you just add stuff to it, and what's happening here is that I've created two local variables, one's called in file, and one's called words.

From that point on it initializes fields, but these two lines right here, forget about them being inside a constructor, there just methods, f open happens to be the function that opens a file. Read lines is this built-in thing that actually takes an entire text file and builds an array where every single entry is populated with one line from the original file.

I'm not gonna show you the file, you can just imagine that the words file that's opened by default is an alphabetically ordered list of all the English words in the language with no intervening white space except for the backslash N's for new lines. Okay. So I synthesize this right here. The read lines that actually preserves the backslash N which is really annoying, but it just does it, so what I do here is I introduce the first ever fields to the lexicon object by saying, you know what, you didn't have anything before, but now you have this words field.

If it had it before it would just reinitialize it, and rebind it to something, but since I'm taking a raw object, I'm actually inserting one more thing into the dictionary that's backing the object, and it's initially set equal to the empty array. Okay. And then I just do brute force for looping. This is what's called an interable, it's actually an array. That's why I synthesize this right here. Okay.

And it just goes through, and it takes whatever words in that words array, truncates off the backslash N and puts it in the words array that's embedded inside the lexicon. Okay. Does that make sense to people? Okay. To whatever degree it's successful for you, just subscribe to your C++ and Java sensibilities as far as what constructors are for and what they're intended to do.

A lot of this is just different syntax. It has some quirks, its dynamics, but it doesn't have a class definition. There's no dot h file, there's no implied interface by a dot java file. It really is just this kind of deal with it as it runs type functionality, but nonetheless this entire thing is responsible for taking a raw object, and building it up to be a logically sound lexicon. If I just show you the next – oops, didn't mean to do that, but I'll bring it back in a second – the next method, it really is a method. It's a function definition that just happens to be – I'm not sure I can get the entire thing in there – the def is over there on the left, but you can just look at this.

Self dot words, there's this bisect function, it's just like B search it's a little bit different. It's more like the lower bound function from C++ in that it just returns either the index of the matching element or the index where the thing could be inserted in order for it to maintain alphabetical ordering. I initialized the words array so that it was alphabetically ordered. All I'm doing here is I'm asking whether or not the word that's explicitly supplied when I invoke this contains word function, whether or not when I do a binary search for it, and get the insertion index for it, whether or not that actual slot in the words array matches the word I passed in.

So I'm actually going to slide this over and see what the double equals is, whether or not it's equal to the word local variable. Okay. You get the gist of what I'm attempting there? Okay. I have some other methods, I'll just name them. They're not that algorithmically interesting. Word contains everything. I'll show you how that works in a second. List all words containing. I'll show you those in a second.

What I want to do is I want to bring my terminal back and show you how this works. Let me make sure I'm in the right directory. I am, and I do in list to make sure I have words. I do. Let me invoke Python. This is how you deal with objects in Python. It's a little quirky, but there's a good narrative that can be made as to why things work the way they do.

If I want to use this lexicon class, just like all the functions inside divisors.py, and all the functions inside the copy module and random module and things like that, I have to import – I'm sorry, from LEX, because this is all stored in lex.py, I want to import the class or the symbol, whatever functionality is associated with the symbol lexicon. Okay. And I do that and it works. I can do this, I don't want to use L, yes I will. I'll spell it out, is equal to lexicon, just like that.

Student:[Inaudible]

Instructor (Jerry Cain):Oh, I'm sorry, well, you can tell me. I didn't do that. Okay. Do this. Wait, can you see it now? Barely.

Student: Yes.

Instructor (Jerry Cain):Okay. Now you can't here me. This right here initializes, there's no new keyword in Python. The name of the class in this case is what's called a callable, and if you invoke it as a function it's a request to build an instance of that class. When I do this, it actually builds EL, I can do a couple things here. Lexicon, if you just type it in, it just reminds you that it is in fact a class. Here's one thing. Just to show you how relevant dictionaries are to the lexicon class, there's a special meta variable that's inside all objects, inside a lot of things, but in a particular object at the moment.

You ask for its dictionary – it looks like gibberish, but you see gestures to a lot of things. You see some weirdly named keys like underscore underscore, module, and underscore underscore init. Okay. Does that make sense? You can clearly see that they're keys in some dictionary the way the way it's scripted out. So the in-memory model of an actual class definition is a list of all the symbols that are embedded inside.

The ones that we created are underscore underscore init, and contains words and list all words containing, and contains all characters or whatever I called it or word contains everything. Okay. Does that make sense? So the actual class object is modeled by a dictionary. That doesn't happen in C or C++ at all. Everything's done in compile time, and it just generates all these 1's and 0's that are consistent with the original definition of the class.

In Java there really is something like this, it's actually not a dictionary, it's a standalone class that gets stored in memory, but it's not that different from this right here, but there really is an in-memory representation of the class idea itself. I can't show you EL; that would be a lot. Well, actually I will show you EL. If I do this, it tells you it's an instance of the lexicon in the LEX module.

When I do this, it's gonna show you the dictionary that's not associated with the lexicon class, but the dictionary that's associated with the instance of that class that I just created. Now this is gonna whiz by, it's gonna take a couple of seconds – actually that wasn't that bad. I can't scroll up because we're dealing with a 150,000 words. Okay. Well, actually I could, but I'm not going to.

What I will do is I will break the – oops, it didn't like that, sorry. Okay. That's a little weird that I did that. Remember that words was a field that I introduced inside, there's no notion of privacy whatsoever in Java. You're only supposed to deal with that by policy, and understand that there probably are fields inside you're not supposed to touch. There is some way to actually mark something as intentionally private using underscores. I'm not saying it's not used; it's kind of a hack.

There really is no enforced encapsulation in Python. It just relies on the programmer to be a good programmer, and to not touch inner fields or functionality that it doesn't think it needs to touch. But now if I do this you won't get the original dictionary, but you'll get that right there. Okay. Before words had this array that went all the way through though zizzaba, and I just happened to empty it out. Okay.

But what I'm illustrating here is that the object itself is backed by a dictionary and all the attributes, in this case there was only the words attribute, okay, but all those attributes are actually keys that are associated with some inner dictionary that's identified by an underscore underscore dict underscore underscore. Okay. Does that make sense? So you see how central the dictionary is to the Python language. Okay. I'm guessing you do. Okay.

So there's that. What else to I want to do. Let's actually just, because it's fun, not because it's really important, let's rebuild the lexicon contains word hello, just to show that it actually works, it does. Okay. And then I wanna list all the words – I love doing this – all the words that have all the vowels. Oops, I didn't do that. That's not – List all words containing – spelling right, yes, okay. Isn't that neat? And it's very brute force. This isn't the magic of Python, that's just they way I implemented it. It's just a regular class.

But the implementation of this is in the last page of the handout I gave out last time, the Python basics handout, and it's just a matter of gleaning syntax. I can tell you right now from experience that when you go out and get a programming job, it very well may be in C++ or Java, it may be in some language you have very little experience with, but you can't be, um, I don't know how to do it. But binary search is mysterious in all other languages. It really isn't.

It's just a matter of learning the other syntax and the idioms that the language supports for getting things like iteration and recursion and classes and structs and objects and all that kind of stuff down. Let's see if there's this. Just that one, and then obviously there's nothing else. Let's see if there's any words that have A, B, C, D, E, and F in them. That's not bad. Okay. N, G. Nope. Okay. Okay. So that's just playground stuff. That's kindergarten stuff just doing that. What I wanna do now is I wanna show you – let me just draw – I'm trying to think if I wanna do some stuff. I do actually. Let me put this to bed.

What I wanna do is make a few remarks about the illusion of objects just being dictionaries. When you do something like this, let's say at the prompt you do O = object. This is the equivalent of java.lang.object. You do that right there. You get absolutely nothing in response to this. You just get O is an instance of the object, but if you do this you actually get that right there. Does that make sense to people?

I'm asking it to identify the collection of attributes that have been accumulated inside the object I'm calling O or the instance of the object I'm calling O. When I do this, and then I do this, and I do this, they're really instructions to insert a new value in the dictionary that is underscore underscore dict underscore underscore. So if I do this, I actually get three fields. I'll assume they hash in this order. Okay.

When you do this you're not respecting capsulation. You're supposed to let methods do this, but I told you already that it doesn't respect any kind of privacy. There really is no support for privacy that's genuine and real. There's that, but doing these three lines right here is like doing this, and in fact it more or less translates to this. And then O dot underscore underscore dict underscore underscore c = [inaudible].

This right here is syntactic sugar. It doesn't look like syntactic sugar, but it is in the case of Python, because this is just taken to be an instruction to not really try to do this, but to do that right there. It's really physically levying an operation against the dictionary that's inside of it. Okay.

So the takeaway point from this – I'll talk a little bit more about this in the first five or ten minutes on Monday, but the takeaway point here is that the objects are backed by these growable, shrinkable containers, they have to be growable and shrinkable in ways that they don't have to be for C, C++, and Java because C, C++ and Java do all this wedding planning up front where everything has to be set in stone before any code executes whatsoever.

Python is a fully dynamic language. Everything's supposed to be able to grow and shrink. Everything about it is supposed to be dynamic, so you're not supposed to impose limits on how big something can be unless it's an implementation limit like you just can't store more than like 2 to the 12th keys in a dictionary or something like that. Okay. So activation record as an idea, it works beautifully for C and C++ where it can actually do all the scrubbing up front, but for Python it can't; it has to back all the objects with a different model, and this is exactly how it does it.

Okay. So you definitely have enough information to crank through Assignment 8. Come Monday, I wanna talk a very little about inheritance. I know you know a little bit of inheritance from either AP Java or from 106A. Okay. If you've taken 108, then everything will be easy. But I will just talk a little about inheritance and how it's relevant to some of the, I think, more interesting ways of doing networking operations which is what this handout I gave out today is all about. Okay. So you guys have a good weekend, and I'll see you Monday.

[End of Audio]

Duration: 49 minutes

ProgrammingParadigms-Lecture26

Instructor (Jerry Cain):Here we go. Hi, everyone. Welcome. I actually have, in spite of what it looks like, three handouts for you today, but we had some photocopier drama, and so I handed out the practice midterm already, and we have two more handouts. The solution to the practice set of problems, and also tomorrow's discussion section handout. I know a lot of you have 107 is falling off the radar because you know the assignment isn't too bad, and the Python's not on the final exam.

But this memorization which was discussed in the Assignment 8 handout is central to you getting the problem done nicely, and if you have any trouble with that then you're gonna wanna go to the discussion section tomorrow because we do an even more difficult example than the one you have to solve in that section handout. We also do a little XML processing and data processing using some of the built-in functionality from Python that I'm gonna go over today. I think it's a cool little section handout.

It's pretty lightweight, but nonetheless, go to the discussion section tomorrow if you have any more curiosity about Python. I'm really regretting not having more time for Python because I think it's such a fun little language. But we've already learned all the paradigms that I wanna go over, so we're just looking at Python as a representative of all three of them. As far as the final exam is concerned, remember it's a week from today. I can't say it enough, but there's no Python on the final, but everything up to the last lecture of Scheme through the entire quarter is fair game.

I certainly will emphasize material not covered in the midterm, but given that the midterm is worth what percentage that it is, and the final's worth a little bit more, I really try to make it so that all of the C material that was emphasized in the first half of the class really contributes about half as much, or I'm sorry, equal weight as all the post pure C stuff. So expect to see one very simple C question, a code gen question that focuses on C++ and references, the stuff I did not emphasize on the midterm, and then scheme, concurrency, etc. Okay. No etc., scheme and concurrency. Okay. Yep?

Student: That was [inaudible]

Instructor (Jerry Cain):Sorry, code generation. So yes. Certainly, yeah. I'm sorry, there's a morning offering, 8:30 it's in Dinkelspiel Auditorium. You're gonna have so much room there because only about 25 percent of you are going to come to that time, and then the afternoon one is in Hewlett 200 which is where I believe your midterm was. Okay. What I wanna do today is I wanna talk about the more modern practices in dealing with XML processing. I think XML is kind of neat. It's not exactly a new technology although it's not exactly young either.

All of you have some exposure to XML processing because you had to do at least a little bit of it, or understand a little bit of it, for Assignments 4 and 6, and when you got to Assignment 6 you really did real XML processing because you were forced to deal with the X-pack parser, this open source [inaudible] that happened to be installed on all our machines here, and you used the X-pack parser to get all the titles out of there, and all the documents, and all the URLs and things like that. What I wanna do today is I wanna show you two different XML parsing models as they're supported by Python.

One of the takeaway points is that as opposed to C and C++, Java and Python, because we're talking about Python, both have built-in functionality for dealing with Internet connectivity, and in particular, XML processing. C and C++ and their specifications were laid down to rest for the most part well before the Internet became all savvy, and well before XML really had a presence in the Internet community.

Java and Python are much younger languages, they're kind of equal age with XML, and so all of them kind of – XML and Python grew up together. There actually is built-in support for XML processing, and Python I wanna use XML processing as our first gesture to Internet programming to see why Internet programming is so much easier or at least more gracefully supported by the Python language. Okay.

So let me just put a dummy fragment of XML up on the board. Let me write it this way. I will write it as entry and I will say address, let me say number is 2210 number and then continuing, and emphasizing the fact that this is all one big stream of text. Street, Hope Lane, this is where I grew up, suburban New Jersey, and close tag, this right there, and then you get the gist, and I'll just end the address part of it, and then finally just open up another tag, and then let your imagination run wild as to what my phone number was.

The reason I'm drawing it this way, is because this is more or less how the text came to you for Assignment 4 and Assignment 6's purposes. Okay. And the reason I'm doing this is as opposed to drawing it as this hyperstructure text tree which I actually will draw it that way in a minute for a different reason, this emphasizes the fact that even though there is structure and an hierarchy to the XML, that the XML parser we're familiar with from Assignment 6 and Assignment 4 actually was a stream-based parser which means it doesn't store it in memory.

I'm sorry, it stores only a constant subset of the entire text stream in memory at any one moment, and as it gets enough text because it sees the end of a start tag or the end of an end tag or it sees some character data, either the end of a stream of it or up to some maximum of characters, you know that it fired little events that we installed into the XML parser. Okay.

So right there, and right there, and right there, all of those arrows point to the place in a character stream where you would have seen some kind of start element tag handler invoked. Okay. And right here, you would have seen some character data handler invoked on your behalf. Right here, you would have seen an end tag handler invoked. Okay. And the stream-based parser that you did or you saw done for you for Assignment 6 is the opposite of functional programming.

It relies on state and side effect to keep track of where you are, and you know that when you read something like this that you're probably inside an address in an entry tag just based on the context of the individual example, and as you leave a number you know that you probably very recently read in an actual number as a text element. Okay.

What Python does, because it's object oriented, and C from Assignments 4 and 6 was not, Python actually uses a more modern approach that I think is a lot more graceful where via two packages from – there's actually a package called urllib, it's old and compared to the other package that has more modern support for more URL functionality, the snazzy name for it is urllib 2, I'm gonna go ahead and import this function called urlopen.

It is basically the equivalent of f open, but you give it not the path to a file name on your hard disk, but you give it a URL, and you just pray, because I don't care about error checking at the moment, that it actually points to a real document somewhere. Okay. From xml.sax import, these two functions make parser, and then a class which I'm gonna talk about called content handler, and then I'm also gonna import sys, all of it because I want to be able to print to standard L. Urlopen is the equivalent of f open. Okay.

It just works beautifully. It's a much more graceful implementation of URL connection, the URL connection class that you're familiar with the two middle assignments in the course. Make parser right here is what's called a factory function. It just promises to hand you back an XML parser that responds to a certain number of methods.

This content handler is actually an interface for – or it's a base class that has to be subclassed, and the subclass has to implement certain methods that actually have functionality associated with them so that it knows what to do when it reads a start tag, and when it reads an end tag, and when it reads character data. Okay. Does that make sense? Okay.

So here is the gist of what I want my XML parser thing to do. I'm really gonna deal with the understanding that some XML document is gonna have a title tag somewhere near the top. This happens to be the title of the entire feed, and it was irrelevant for the purposes of Assignment 4 and Assignment 6. I was interested in the title of the actual articles. There's actually a tag called channel. Inside you see things called items, and inside that, you see titles something like that.

There is a link and there was a description. I don't care about those. And you would see the same thing several times. Okay. Ends channel, ends – that's interesting. What I want to do is I want to write enough of a program that actually can read a URL, assume it's a URL that leads you to one of the RSS feeds that you were dealing with in earlier assignments, and I want to go in an extract all of the titles and just print from the standard out.

So this as content, and the same things in other item tags are of interest to me. I want to use a stream-based parser to do it, and I want to do it in Python. Okay. Just assume that content handler knows how to detect whether or not it's inside an item tag or a title tag within an item tag or it's leaving one or it's actually dealing with character tags.

There'll be more this, and this is really the heart of the code right here, but what I really want to happen is I wanna define a function called list feed titles, and I'm just gonna give it a URL, or I'm just assuming the URL is in fact something like Washington Post, anything you dealt with from Assignment 6. We're gonna use blog RSS feeds because they're campier, but I will just use those as an example. What I want to do is I want to not bother with error checking, I wanna call urlopen.

That basically gives me the file star that knows how to extract text from some remote document at that URL. Parser is equal to make parser. Now there's a little bit of mystery as to what that's doing. You have to just assume that it's some XML parser, and that this make parser thing is just returning a generic XML parser that doesn't know what type of XML document it's gonna be processing. Okay. But you know enough about XML that they all kind of have the same idea.

They're textural, but they're hyperstructured. You don't know the tag names that are relevant to any one particular RSS feed, but we're gonna make this particular parser relevant to RSS news feeds. So it's gonna eventually be able to detect things like channel and item and title within item. Okay. There's one method called set content handler – I'm just gonna do this.

I haven't devised this class yet, but I can tell you at this moment that I'm just gonna say RSS handler is gonna be a subclass of this thing called content handler, and from a generic standpoint, all you know is that by installing an instance of this type of class here, that we're gonna implant functionality in this class that knows how to take a document of this structure right here, and just print out all the titles. Does that make sense? Okay.

So now the parser has gone from generic to RSS news feed specific. Then I'm gonna do this: parser.parse info. So the only two methods that you need to rely on being available to your parser is the set content handler. You don't have to call it up, but then your parser wouldn't do very much, and the parse method which is kind of the method you would expect to send to your parser object. Okay. If I don't call this, the thing that's installed by default is an instance of one of these things. Let me just show you what the implementation of content handler looks like.

Class, I haven't actually looked at the implementation behind the scenes, but it behaves more or less like this. There are five methods that are of interest, but I'm only gonna focus on three of them. There's an init method certainly, but I'm most concerned about this, start element takes self, it takes tag, and it takes attributes. There's another method called end element that takes self, it takes the tag that's being ended, and that's it.

It also has a method called characters that takes the self which is equivalent of this pointer, it also takes data, and that's the signature for that. There's also a start document and an end document method. I'm not worried about those because I don't have to do anything specific for those. There's also a default constructor. Okay.

When you deal with characters not surprisingly this gets invoked anytime the parser is parsing, and it's halfway through something that's clearly textual – not halfway through, either entirely the way through or the character data, the stream of text that is there is so long that it really just doesn't wanna build an arbitrarily large string, so it goes up to some max and hands the character data over to the first of what will be several character method indications. Does that make sense to people? Okay.

When it gets to something like that, it would invoke the start element method, passing address as tag, and attrs, in this case attributes would be an empty dictionary. This is supposed to collect all of the things that are passed as arguments to a tag. You normally see that more in HTML. You certainly see it in XML as well, but when you see a tag like this – hold on, I'm not doing very well. Let's just do that right there, and maybe you have a couple of other things. And you know what I'm structurally getting at there.

This happens to be an HML anchor tag where the tag name is A. Those three attributes, one clearly is the h ref that's associated with the Google URL. This, that, and that would be stored as three keys in a dictionary. That, that, and that would be the value that's attached to those keys, and they would be all bundled in a Python dictionary that passed right there. Does that make sense? Now we don't have to worry about that in an example, and we also don't have to worry about it in the way I'm gonna be pulling the RSS news feeds. Okay.

This does the same thing in response to something like that. It actually passes the tag name to it so it knows which particular tag is ending. Okay. You should be able to infer it by a clean XML document because if some tag is ending, it's supposed to be the one that most recently started that hasn't ended already, but nonetheless this is invoked on your behalf because you may want to do some special processing when you notice that you're really at the end. Okay.

Now if I didn't have this line right here, that would be the class that handles all of the XML as it's pull through in this stream-based manner. The implementation of all three of these things is the equivalent of this. Now that isn't real Python, but you know what I mean when I put that there. Okay.

Well, the [inaudible] isn't supposed to use curly braces because you're not supposed to use curly braces at all in Python. Okay. At least not as block delineators, so the keyword that's used in Python is the word pass which basically is a placeholder which means I am nothing, but I'm functioning entirely as a no-op implementation. Okay.

What I wanna do, I wanna keep that there, is I want to, just by protocol you normally subclass that there so that you get the full suite of methods that should be available to a content handler by default, but you displace the ones that aren't good enough, and certainly start element, and end element, and characters actually have to do some kind of bookkeeping, and in some cases some printing.

If I really want to print that, and that out, that's certainly not gonna do it. It really would in this context, if I didn't have this third of the four lines right here, it would really call those four methods of those three methods up there, but all of them would be like, okay, you called me, great, I'm just returning. Okay.

What I wanna do is I wanna print out the character data that's inside every single title tag that's inside every single item tag, and I say it that way because I do not want to print out this title up here. Okay. Make sense? So what I wanna do is I wanna implement this thing called RSS handler. I would name it a little bit better if I had more space on the board, but I didn't so I used standard RSS handler.

And I want to design it as a class where it keeps just enough information to know whether or not it's inside an item tag, whether it's inside an item tag and inside a title tag, whether it's inside a title tag but not inside a title tag. Things like that. Okay. Because we're gonna use some Booleans that are tracked by the content handler class to figure out whether or not the character data that's actually being passed every single time to the character's method should incidentally be printed. Okay.

There's no way to control when characters gets called. It always gets called. It's gonna get called right there and right there. It would get called right there, and for all of the other things that are textual, but I only wanna print out these right there. Does that make sense? Yes, no? Okay. So the way you subclass in Python – actually I wish I had like three weeks to talk about inheritance in all the different languages because it's really, really interesting stuff, but I'll try to compress all the parts from three weeks of lectures into like two or three minutes.

To build a subclass in Python, you just define a class – you know what – class RSS handler, and you do this, you just say – and this right here is the equivalent of extends from Java. Okay. In the new version of Python, it's actually the 2.6 version of Python, I think it's 2.5 or 2.6, I forget, everything subclasses that object type I showed you on Friday by default in the same way that most classes by default – I'm sorry, all classes by default subclass java.lang.object in Java unless you specify which class it's subclassing.

I have to do the same thing here. If I didn't provide this it would just subclass the built-in object object, but I want it to subclass that so it has implementations of those three methods already. Okay. Does that make sense? I have to do a few things. I do have to define a constructor. The reason I do will become clear in a second. What I have to do here is I have to call content handler.

This is how you basically do the equivalent of a super call in Java. You have to say since I'm subclassing content handler, I have no idea what's involved in the construction of the pure content handler portion of it, but I should just make sure I call it in case there is something relevant there. But then what I want to do is I want to introduce to Booleans, just on the fly, it's just the Python way, in title is equal to false, and then all the sudden it's fairly clear I think what those Booleans are gonna be doing.

The double underscore at the front is a gesture to the Python environment that these are supposed to be treated as private variables. It's not formally enforced. All it really does is it actually mangles the names of those private fields to be something else. We can internally refer to these things right here, but it actually changes the variable names in the memory model so that if you try to reference these things from outside the class you have a harder time doing it.

It's not encapsulation, it's actually more of a hack than that. But nonetheless this is just convention for dealing with privacy in Python classes. As far as the characters data, the characters method is the easiest of the three to implement because there's no tricks. Character where I pass in self and I pass in data, normally I would just do this, sys.std.out.write, is the equivalent of print f. I would just print the data.

But do you recognize that that would print all of the character data in the order that it comes in the document, but nonetheless I don't want that. Okay. So what you need to do is you have to assume that in title will be set to true when we really are inside a title tag because then if the character method is being invoked, it's title text data as opposed to arbitrary text data. Okay. Does that make sense?

So the start element, and the end element, that's more about maintaining these two Booleans than anything else. Does that make sense? Okay. So I have two different scenarios that I want to dispatch against inside start element. I have self, I have a tag, which is a string that's going to be like title, or link or channel, or title or whatever, and then I have attributes which are being ignored here.

And I have two different if tests that I wanna worry about. If it is the case that tag double equals item – you can compare strings, you can double equals like that – that means with a normal RSS feed, you're actually entering an item subdocument where you expect to find a title, and a link, and a guide and a start data and a description and all that stuff that you're familiar with from Assignment 4 and 6.

If that is the case, all I wanna do is I wanna set self in item to be equal to true. Okay. You don't have to go on the next line if you only have a one-line id block. Otherwise, if tag is equal to title – the way I'm doing this is actually incorrect, but let me do this much – and I'm assuming that in title is being set equal to true because I've very recently stepped into not only an item tag, but also a title tag. Okay. However, this is not quite the right test.

You are guaranteed to only have one level of items tags in an RSS news feed, but title actually comes at two different levels. Now you could argue that that was silly, they could have used any one of a million different strings so they didn't have to overload the meaning of title in RSS, but I didn't design it, so I have to deal with their design. So if the tag is equal to title, and incidentally end is really the key word, and it's also the case then I'm in an item, then go ahead and catalog the underscore underscore in title to be true. Okay.

So do you understand how start element – I don't wanna say it's sophisticated, but it's not trivial either because you really have all these little side effects associated with start element calls where the two Booleans inside are constantly tickering and toggling betweens trues and falses, and technically you need both of them to be true, although we're only gonna check in title to be true. Technically both have to be true if you're really gonna be printing out title data. Okay.

As far as the end element is concerned, self-tag, if it is the case that tag double equals title, and self in title, then I know that I very recently responded to some character methods that printed some stuff out. Does that all make sense to people? This right here does not print a new line, so if I really want to publish the title on its own line, then kind of is a little last gesture to the title that just got recently printed, I will do this. Okay.

And I will also – I will set self in title to be equal to false. If it's the case the tag double equals item, then I will go ahead and not print anything, and do that right there. Okay. So the start and element tags are all about taking falses to trues and trues to falses, and the character method is incidentally printing character data when both of the Booleans are true. Okay. Do you understand what I mean when I say that's the opposite of functional programming? It's all about side effects. Okay.

So just so that you know that this is not really theory, and that it really works, let me show you three ways to examine an RSS feed. I'm not saying that you should be reading this blog, and I'm not endorsing it at all, but hopefully it'll – bring this up – somehow it's the case that when you do www.gizmodo.com you get a blog about cool and trendy gadgets about people who are often very bitter. Okay. But this is the presentation, and you can see just be scanning through all this stuff that there's titles, there's content, there's probably stuff that's pulled from URLs.

I can't really show the entire web page here, but when you visit gizmodo.com – I don't know how it constructs the Web page, I would not be surprised if it constructs the Web page from its very own RSS feeds. Okay. It might not do that because the Web page structure doesn't change all that often because the RSS feeds don't change all that often.

So in theory they could build them from the RSS news feeds, but they might just build the HTML document once, and then use the same static HTML document until either a new article comes along or the comments are updated or something like that. Okay. What you may not know is that you can just look at the raw RSS document, and this really is Firefox 3 beta's way of displaying an RSS feed as if it was an HTML page. It has a slightly different presentation.

This is the type of presentation that Apple gives to its – I'm sorry, Firefox gives to all RSS news feeds, and it looks more apple.com ish in its default presentation. So there's that. I actually identified the URL of the RSS news feed to the Web browser, and those are the types of things you dealt with within Assignments 4 and 6. Here's another way to actually look at the content.

This is the least – I think it's the most informative from a networking standpoint. You're not gonna be able to really digest what just happens here, but you understand in both cases when I actually viewed the full gizmodo Website, and also when I viewed the RSS news feed that my computer is talking to some other server. I know you know that much. The way they communicate is they actually pass text back and forth to one another, and the text somehow encodes what document is requested, and what the document really is.

It's not a very long conversation, in fact, my computer speaks once to the gizmodo server, gizmodo hears what I say, and in response actually publishes a much longer stream of text back to my computer which is either gonna be the RSS or the HTML. So what I wanna do here is in two phases – is this everything? No it is not. So now I'm realizing that you didn't see the whole Web page did you? There's the URL for the RSS news feed. Okay. Better.

Okay, so what I wanna do here is I wanna bring this back. I want to telnet to – let me see what the – feeds.gawker.com – before I press enter, let's copy this thing. This is gonna look like really random stuff. Fortunately the response is not gonna be very long. Let me scroll back up to show you what I type in here. Right there. You see all of that? Okay.

This line right here, basically telnet is an instruction to pick up the phone, feed.gawker.com tells you which house you're calling, and 80 tells you which of the 2 to the 16th different phone numbers you're calling. Okay. And the HTTP server that's running at feeds.gawker.com is listening on a port called 80 for HTTP requests. My HTTP request happens to be structured this way right here. Okay. You probably heard of things like get and post just as verbs that come up in programming. Okay.

This is just an instruction, when I'm speaking to the phone after they answer, and they answer right here. It says so. It connected. I'm interested in getting this document, and this is the actual protocol. The more modern protocol is 1.1, but I didn't know for a fact that it would actually be implementing that, so I actually went with the one that's been around for several years, and in response when it hears that, and basically the equivalent of a period is two carriage returns, it comes back with that, and a little bit more.

You know enough about – I know you have digested HTTP response code of 302 from Assignment 6 where that involved the redirection. Okay. So what this is saying is that thank you very much I have it, but I want you to actually consult this URL to get the RSS feed instead. So what I'm gonna do is I'm gonna say, okay, that's fine, I'll call these people, feedproxy.feedburner.com, and I will ask for this document instead, and be very fast about it because otherwise it hangs up the phone, thinks it's a prank call, and that should be enough.

It takes a few seconds unless I messed up. Actually this would not be good. Well, I don't know what's up. I didn't type it incorrectly did I? Oh that's disappointing. Let's try it again. It's really stuck. Whew, that's not good. I don't know what's going on because it won't hang up the phone. Okay, well, anyway, I guess I lose. Here is the response. Okay. It comes back with this, and it's hard to get a lot out of that, but I can tell you right now that the thing I'm interested in, let me see if I can find it, there's the title that I'm trying to ignore up there.

There should be an item somewhere. There's just you see all these things that you're familiar with. I don't see an item. Hold on a second. Oh, there's one. That's good. Okay. You can see that right there. Okay. So that's the one tag, and it actually spits back to the computer this stream of text that happens to have new lines in it so it looks like a human-readable document. Okay. It would have worked out just fine if it didn't have all these new lines inside of it. And that's the text that comes back to me in response to say a URL open call.

And I actually say thanks for that text, I'm gonna hook this in file URL open result to the parser, and I'm gonna call parse against it, and it's gonna pull all this text in, start element, start element, start element, characters, end element, cart, that kind of thing is gonna happen. Okay. Based on the text that's been read in. Okay. Does that make sense to people? Okay. So that worked.

The advantage of the type of parsing I['m discussing right here is that some XML documents, and not this one, it's not really that large, but in principal XML documents can be as large as they wanna be. We're probably dealing with a few kilobytes of data here, but there's nothing to say that you can't have an XML representation of like the English dictionary or like all world languages or something like that. Okay.

There might be an XML document that stores all of like the books written between 1700 and 1800. That's gonna be more than a few kilobytes, but the advantage of this type of parser, and the one you're most familiar with now, is that it doesn't try to store the entire document in memory.

It reads in jut enough to make some progress and as long as you can deal with the XML document in it's read-only capacity, and you only need to make one pass over it, you can actually digest the entire thing in small pieces, and discard what you've read in, after you've actually processed it to the degree you need to. We happen to be using the content handler subclass in Python to get the job done. Does that make sense? Okay.

That isn't the only way you can process XML documents, and the way that was not emphasized in any of the assignments and not in this example, is that there are some XML enthusiasts who for their reasons, actually like to put the entire XML document in memory. Now there's an advantage to that because if you can come up with an in-memory model of an entire XML document, you can do more than just print it.

You can reverse it or you can snip parts out, and you can sanitize parts of it, or you can restructure it, or add new items dynamically into the tree in ways you can't do if you're using the stream based read-only one pass approach. Does that make sense to everybody? Okay. So there was a second model of XML processing. This SAX model – SAX stands for Simple API for XML. Very popular low memory imprint. It's a great way – like I have an opinion – it's actually a common way to process XML documents.

Now what I'm gonna do is I'm gonna punt on this. I'm not gonna be able to get this entire example – I don't think I'm going to. Go ahead, I'm sorry. I'm gonna draw the XML document a little differently. Okay. This is an important way of digesting, and understanding how XML can be processed because this is really how web browsers digest, and manage, and render HTML. Okay.

If Web pages used a stream based approach to building the Web page after running the HTML, and it discarded everything, there would be nothing dynamic about Web pages at all, except for the fact that it could be constructed dynamically, but there'd be no interactivity at all.

You know how like when you press a button, like you're shopping at Amazon or something like that, and then the page updates and your shopping cart gets one more item in it, that's because the in memory model of the HTML document is actually updated by that button click. So you actually update the HTML document, not the original one, but the one that's being managed by your browser. Okay. You can do something similar with XML.

My first example, what I'm motivated toward isn't really a dynamic example in that it's gonna change the tree, but I certainly could if I wanted to. Let me just draw an item, subportion item, you gotta be careful about this – there's title, hello there, end item, there is a link ACTP, blah, blah, blah, and then there's a description tag. I'll abbreviate it that way.

It's what it was, and then there's the end of the item. Okay. That's a subtree of the overall XML tree that is of interest to me for this current example, and you can imagine that if there are a stream of these things, one after the other, effectively a stream representation of an array of item objects. Okay. They're surrounded by this thing I'll call channel. That gives this very left to right view of an XML document. Okay.

The overall thing has one tag up here called RSS, the end tag is RSS right here, but the root of the entire document which is managed by this node right here, is on the left of the drawing as opposed to the top of the drawing. But you've heard about XML documents. You may not have had too much practice with them, but I know you've at least seen XML documents before.

What some XML processing models go with is a tree representation of the XML document. Here's an actual node that has the tag RSS inside of it. Okay. It has several children which themselves have several children, and this may actually have as one of its children a channel node. Okay. So do you understand how I've basically taken this as a document? I've made sure that and that are associated.

It's like these are like two ropes at the end of a hammock and I've brought them together, and of course the hammock's lying on the ground unless I actually rotate up like I am over here. Okay. This channel would have several item nodes. The items themselves would have title, and link, and description, and even link actually is the same type of node. It actually has a little node down there.

I'm drawing it differently because it's actually a text node that has associated with it the actual text that's inside of it. Does that make sense? Now that isn't a very sophisticated drawing, it looks a little scary, like the worst mobile ever, but you can imagine how that will translate to an end area tree in memory. Okay. XML, as a standard, every single start tag is supposed to be closed by an end tag. There's much more structure with an XML document than there is even with HTML.

It turns out parsing HTML is a more difficult problem than parsing XML because XML has much more required structure to it. You can imagine something like this existing in memory where this and this and this and this are – and actually all five of this rectangles are categorized by a struct or a class that's just called node.

These four right here are specific types of nodes and C, C++ or Python, they would be modeled as subclasses of nodes that I would call element, that's what Python calls them anyway. Elements emphasize the fact that they correspond to things that are delimited by these greater than or less than signs. This would be a peer subclass of element called text node or called text in Python that actually carries whatever text is associated with something like this or that or that. Okay. Does that make sense?

There is another package in Python that's dedicated to the DOM model of XML processing. DOM stands for Document Object Model. Object is clearly there for, I think, obvious reasons, but it has this object-oriented approach of modeling an entire XML document, that's why the D is in DOM. Okay. And then if you build something like this in memory, you have an entire in-memory tree that represents the XML document, you can look for all of the nodes that have item inside them. Okay.

Then for all of those you can look underneath all of those subtrees for the item, for the node that have title inside of them. I know that all of those title nodes actually have one child that are text nodes that have text associated with them. Does that make sense? Okay. Conceptually, I think it's easy to digest.

It is a little bit of work to understand what all the methods in Python or Java or Ruby or any modern language that has Web capability built into it can do, but if I wanted to construct an RSS news feed, what I could do is I could collect all of my titles, and maybe I have normal C structs with title fields, and link fields and things like that. I could actually dynamically insert and construct all of these things. Okay.

And once I construct this, I can serialize it, traverse it in this in-order matter like you're very familiar with doing just regular binary search trees that way. You can do it in reversal of this, know how to print on behalf of this you print a start tag, you may print any attributes that are associated with it.

You fully marshal all of this in sequence then when you come back to here, you print the end tag associated with that, and if you follow that recursive formula throughout the entire thing, you can construct an in-memory model of your RSS news feed and then serialize it, do some kind of to print function or something like that. Okay.

That type of functionality is certainly built in to Java, which you probably do not hit on in 106A, but you certainly will hit on it in 108. It also exists in Python. Okay. Does that make sense? Now there's one example in the handout. I will try to get to it in the first few minutes of Wednesday's lecture. As far as Wednesday goes, I'm not sure. I do wanna bring you in because I do wanna talk about some other languages.

I'm trying to coax a very smart, but very young, and very nervous colleague of mine who is a little worried about speaking in front of people his age as an authority in the language, but he's super smart, smarter than I ever will be, and certainly was when I was 22 years old. He knows this language called Haskell really, really well.

He also knows the language that I kind of know pretty well called ML which is like Scheme, but with data typing. Okay. I certainly wanna talk about those, so what is going to be presented to you on Wednesday is really a mystery, but there will be something. So by all means, come on Wednesday. Remember the section tomorrow, and you have an assignment due Thursday night. Okay.

[End of Audio]

Duration: 50 minutes

ProgrammingParadigms-Lecture27

Instructor (Jerry Cain):– cell phone off. Hey, everyone, welcome. I don't have any handouts for you today. You should pretty much be done with all the handouts. Yesterday's section solution hasn't been posted on the web yet, but Ryan will do that now. And the next time you are going to see me will be Monday morning at 8:30 or Monday afternoon at 3:30.

Remember you can take the final at either time. You don't have to tell me ahead of time which time you are planning on taking it. You can only take it once. But I will post the exam as a handout at 3:30 that afternoon, so SCPD students who are watching me right now just plan on downloading it. I am a little more flexible on when you take the exam because I know you work, but I need the exam faxed back from you, faxed in by Tuesday at 5:00 because we are gonna crank on grading that Tuesday night.

You have an assignment due tomorrow night. You can use as many late days as you want to. It was designed so that you weren't supposed to use five late days on it, but if you have five late days and you want to consume them and hand it in after the final exam that's not a problem because we are not going to be able to grade that until after the final anyway.

You should definitely get – if you haven't got Assignment 4 back yet, then let me know because you should have got those back. I have seen all the grades fly through my email. Assignment 6 is being graded right now. I told my TA's I have to have that back. It's a seashell. They have to have the Assignment 6 back to you by the weekend so you can make sure that you understand shredding to the extent that I am going to test it.

Assignment 7 I actually don't think you are going to get back by the final. But that was the Scheme assignment. Historically, if there are 200 of you in the class, 195 of you get A's or A-'s on that because you all get it working and it's this new language so we actually have – I don't want to say we have low standards, but we just don't press you on style that much.

But if you've got that working, then you are probably fine, so you are not going to be surprised by anything in terms of your feedback on that assignment. Okay, so I left you with enough Python to get the assignment done. What I did is I invited a coworker of mine from Facebook who is responsible for that noise. No, I invited a coworker of mine.

He was actually kind of my boss for the first six or seven weeks of the time I was there, and he is a programming language enthusiast as well. He knows two languages called Haskell and ML that I don't know as much about. And I asked him to speak about one of them called Haskell.

So I am just going to hand it over to him and let him kind of present a language to you that you would not have otherwise seen had I been the only one teaching the class, okay.

So let us do a little bit of a microphone switch right here. This is Sasha Rush and I will do this. And I will let him talk.

Guest Instructor (Sasha Rush):Okay, so I am gonna talk today about Haskell, which is kind of like an avant-garde programming language. It's not something you will see in a lot of companies using these days. If you want to get a job, you should be a really good C programmer. If you want to program the future you should learn a lot of Haskell. Think of it like as like the Schaumburg of programming languages.

So here is a little history of Haskell. It comes from the same family as like Wisp and Scheme. It's a functional language which means setting variables is discouraged. Using functions all the time for everything is really how you got stuff done. There is things like map that you will be used to from Scheme but there is like a million less parenthesis, so that's really good.

Then that kind of went down to these languages called ML and there is a language called Ocaml which is like a modern version of ML that is pretty popular. These languages took Scheme and kind of merged it with C in an interesting way in that they are statically typed. So you have integers that are actually integers and if you to try to like use an integer with a string it will fail.

That's really nice because you add a lot of speed to the program but it's really annoying in like C because you have to type integer everywhere. So the kind of brilliant idea behind ML is you write code that looks like Python code, but it figures out the type for you and so it will throw an error but you don't have to put any of the definitions down. It's like pretty amazing that is brilliant idea that was invented like 30 years ago isn't used today, but I think more and more it will become pretty standard.

The latest version of C Sharp and future versions of Java Script will possibly have this kind of stuff in it. So then this language came out called Miranda, which was like a proprietary language, but had this really neat idea that we will talk about a little bit later which is that it had this thing called lazy evaluation. And lazy evaluation you won't see in pretty much any language but Haskell.

The reason of that is all the researchers working on this problem got together and they said, "Hey let's make kind of an open source version of Miranda that's lazy and is standard and we can all work on it together." So, it's a really good thing if you want researchers to look at your language don't make it proprietary.

This all came together in Haskell 98 which, as per the name, came out in 2003. And that's kind of like the current modern version of Haskell. Since it came out in 2003, it's been incredibly popular with a small group of pretty dedicated people. And they have been kind of working on it pretty hard core for the last five years. It's getting to the point now where it's like really good.

Okay, so Haskell is pretty neat. So it's safe like Java. So when I say safe I mean it's really hard to make a program that compiles and then fails. So this is like really knowing about a language like Python is that you are constantly running your programs, they're constantly failing, you are constantly writing them again. It's really nice because, particularly for applications that matter, things that aren't websites that the program that you compile you know it's not going to fail for like a dumb reason that you like left something in.

So what does this mean? Well, this means if you try to add a string and an integer it's going to fail. The little colon syntax is kind of like the cons operator in Scheme. It means like put this at the start of a list, so if you try to put a string in a list with numbers it's also gonna fail. So this again is different than Scheme because Scheme is not statically typed. Scheme will let you put anything on the list.

In Haskell, if you have a list it has to all be the same kind of type of thing. Okay, so it's like Java, but safer. There a lot of kind of funny issues in Java. Here's one of them, so, I have an object called temp and then I set it to null. Then I try to run this method that I like and know it should be on object, like it's guaranteed for me to be on object, but it can fail because temp is null.

So like this is a pretty common thing that most people are used to in [inaudible] languages but it's this huge issue like there's all the possible run time errors that like you are not protecting for. Here's another one. so we have this guy named object, I'm sorry, named temp that was an object, and I am trying to task it to become a string which is okay in Java. You're allowed to – I think it's called downcast. You're allowed to move from one guy to its child.

And it has this huge possibility of failing because in this case it's not a string; you can't pretend it's a string. Okay, it's expressive like Scheme, so like I said before, so you have this map function. So the kind of square brackets are a list. It's kind of a list literal. It's similar to the list literals in Python and you can run kind of anonymous functions over it.

So this function maps adding one to a list and you get back a list with all the numbers plus one. Here's another example, this is a sort by function. Sort by takes a comparison function and does a sort with that function, so this one here just sorts a list by the standard ordering.

Okay, it's fast like C. It's not as fast as C but it's getting there. It's kind of like on the order of magnitude and here are like some stats. This is not the way to think about speed of languages but it gives us an easy way to look at it. So this is from the computer language benchmark game online. Just to give you an idea, the big difference is that types not only guarantee like safety.

They also allow you to make the language pretty fast. You know that when you see a plus sign that you are going to be adding like two integers and not going to have to check for this failure case, and that makes the language a lot faster.

Okay, most importantly, it's pretty fun. When I say fun, I mean like if you want to do a multiplication function you just do a multiplication function. You don't have to like write all other stuff around there. You can do things like mixed arrays. This is kind of like the tuple syntax in Python, I don't know if you have seen that in this class.

And finally, it's got a lot of these like crazy things you can kind of just throw in. Jerry said that you guys saw very briefly list comprehensions in Python, so list comprehensions were idea taken from Haskell. This is a syntax for taking all the numbers from one to ten, multiplying by a two and giving that back as a list. So, this guy on the right here is kind of like range in Python and the whole thing is like a map.

Okay, so I should say now that I am going to show a lot of code examples as we go through Haskell, so feel free to interrupt me at anytime and ask questions. I wasn't really exactly sure at what level I would be doing this at, so some of this maybe like a little bit crazy.

Okay, so we are going to look at Fibonacci, which is like pretty standard, I guess, programming example. So, for you guys who don't remember it, the Fibonacci sequence is simply the sequence where you take the last two numbers, you add them together and you keep on going and going and going and going. So, this is kind of a definition.

Okay, so, here's the Fibonacci sequence in Java. So we have a function fib. We give it an N, which is how long we went the sequence to be. It returns a list of integers. Java is really nice. It has this new generic syntax that lets you say what the list is of. So the first thing we do is we declare. We declare the list. That's our sequence right there, we set the first value, we set the second value, and we do a for loop.

It's pretty simple. It kind of describes the program that we are doing step by step, each line says what it is doing, does something, and moves on to the next line. So I think that's pretty good. So this isn't in Haskell. So where did it all go? Where's all the coding, right? Where's all the writing?

So in Haskell we take a different approach. We go back to the definition of what this thing was and we like just write it, okay. We are not concerned with how the machine represents it, we're not concerned with the ordering, how it's presented to the user, we are just concerned with, like, what's the math? Like, what's going on here?

And you can kind of describe it in the most basic of ways. And so I told you that this is safe, I told you that this is fast. So you can see that it's fun but like the other two, it's hard to describe what is actually going on here that makes this work. So, let's dive down into the details.

Okay, so, here are the two bodies of these functions. So let's try to map like each part to each part. So I said before that the colon means a list operator. It means like hey, put this at the start of the list. So we can see that we started off with the one in our list. So that maps down to hey, this sequence of zero is equal to one. Then we can see the second part is equal to one. So that part, I think, makes sense.

The part I am going to have to sell you on is what this last guy does. So, what's going on here? So one thing that you can see is that we are using the word fib within that expression and we are also setting fib. So it seems really weird that you can use the variable that you are setting within the variable itself. So that's a little bit strange to get over.

So the reason that this works is that we want to think of this not like a variable but almost like a function. So it's like a recursive function that you have probably seen before. So if we think of fib as a function it makes a little bit more sense, we are taking like the old value of fib and putting within the use of it. What does tail mean? Tail is similar to, I guess, like [inaudible] in Scheme or tail in Scheme. Yes, CDR it means take the end of a list not the first owner.

And what is this zip with doing? Zip with is kind of like the map function. It says, "Hey, map this function over this list." Except instead of just mapping the function over one list it takes two lists and combines them with the function. Okay, so, we take the two lists, we combine them and we are good. So, let's look at this in more detail. So, first of all, how can we use fib as a variable?

Well, what do we know about fib? We know that fib starts with one-one. It's a list that starts with those two values. And after one-one it's a question mark. We don't know anything about what's going on after that. Okay. What do we know about tail fib? Well, we know it starts with one and then we don't know anything about it. Right? It's the same blankness that we saw before.

So then the question is what is the zip with guy? So as I said before it's going to add these two guys together. So we know we have won one and we know we have the tail, so we zip them together and we get two. Once we know that's two that goes to the end of the other two lists. So we have it there. We then add these together again, we get three and we keep on going.

So, what happened there? So the trick is that Haskell is what's called a lazily evaluated language. And that means it will do no work at all until it has to. It's like the undergrad of languages. So, we start with one-one and then we go through like zip with. It's all in what's like this blocked. It's like in a jail. It's called like a funk.

It's something that's there that we have to evaluate that we haven't evaluated yet. So then I say, "Hey, I want to take the first five values of this guy and I want to get them out." So then what happens is the first five values get kind of like pushed through, they get converted from like funk land to like re4al values. And then we just leave the rest. We just say, "Hey, we are going to evaluate that sometime in the future."

And that’s pretty cool. It’s cool in that it means that we can kind of have infinite lists. This list can go on forever and it’s totally cool in your program that the list is there because you are not going to spend infinite time evaluating the entire list all the way through. A good comparison like this is the X range function in Python.

So instead of actually constructing the range of the entire list it simply just takes one value at a time as you request them. So it’s kind of like that except for everything, so that’s like that’s the default standard. So, here’s a good example, well let’s see, let’s skip this guy.

So, one thing that’s kind of frustrating about programming languages is that if you want to have a function that kind of works like the if statement does it’s really hard to write it. So imagine we have a function in Python called my it. And you take three values, say like Boolean value of like what branch to go to and the if branch and the out branch.

And then if the first guy is true we take – oh, sorry this is a typo. If the first guy is true then we take the B branch. If the second guy is true – if the first guy is false, we take the C branch. Sorry, totally out there. So, why can’t we do this in Python? Anyone have any ideas? So, what happens if the B value is like print hello? So, if you have any kind of like effects in either your then branch or your else branch, like print, for instance, or like, I don’t know, you make like an internet request, then you have this problem where if that branch doesn’t get called that value still goes out on the screen.

Okay, so if you say like if – if true print hello, if false print goodbye, right. You’re gonna get hello goodbye because both those statements are gonna get evaluated before they are sent to the function, and then the function is just going to return the results, okay.

So in a language that’s lazy nothing gets evaluated, it just remains in its funk until it’s needed. And then when it’s needed, it gets evaluated. So the branch that failed would just go away. Any questions with that? Okay.

So then the second question is well how can it be fast? I said before that to be fast you really need to know the types of what’s going on. And if you look at our old guy we didn’t write any types at all. There are like no types. There’s no ints, there’s no functions, there’s no lists, there’s no anything. So the trick is that Haskell, it kind of like figures out the types.

So I am not going to go into this too much. There’s like a whole body of study on how to do this correctly, but we can kind of just play with it. So, what types do we know to begin with in this function? Anyone? What values here have types? Yeah, the one and the one; we know these guys are both ints.

Okay, and what’s the type of fib, what did I tell you about lists that we know about the type of fib? What’s that?

Student: Homogeneous.

Guest Instructor (Sasha Rush):Homogenous, right. Okay, so if the ints are both there, okay. I'll stop asking questions, but if the ints are both – if we know that they're both ints and we know they are in a list, that means we know that fib must be a list of ints, right? Off hand, even to start with? Okay, so if fib is a list of ints it means this last guy has to be a list of ints or we will fail, okay. So let's check that it is.

Well, we know that fib is a list of ints because we just said so and we know that tail fib also must be a list of ints because it's just the tail of a list of ints, and we know that zip takes two lists and adds them together, so therefore it's also going to be a list of ints. So that means the whole thing type checks and we're good. So the complier knows what the types are, it can make it faster. We know that we didn't make a mistake, so it's safe and everyone is happy.

Okay, cool. So now let's go kind of more deeply into what the types are within Haskell and look at them. Okay, so we are gonna start off with a bunch of basic types. We're gonna have an integer type, we're gonna have a float type, we're gonna have a character type.

Okay, and like Scheme, functions themselves are also gonna be a type so we can pass functions around in that way. So the way we look at types of functions is with the kind of double colon notation. Think of the double colon notation like a declaration in Java that just says this is what the function is. So, what did it say?

Well, we have a function called add one. It takes a number and it adds one to that number. You don't need to use the word return at all in Haskell, so the result it's returning is the number plus one. So the way that we write it is we write int arrow int. That means we take an int and we return an int.

Okay, how did you do adding up two numbers? You write the word add, you write the arguments as spaces after the function. So it is add, val one, val two and we just add those guys together. Now this guy is a little bit confusing, the way you write the type signatures when you have two arguments is arg one, arg two, return value, okay. So that's just how we write functions.

Okay, so there's another kind of type and this is the type like the user defines. So in Haskell you are allowed to define your on types and kind of just put them into the system. Here is the data type for Boolean. So a Boolean type can be either true or it can be false, okay. We put that guy in there. It's now a type in the system. We can write the word true it means something. We can write the word false it means something.

Here's how characters are represented. Characters is just an or with all the different letters possible. Here's color, so color can just be like red, green, blue, whatever. You can kind of write any of these that you want, just like anything you make up you can just put into the language itself.

Okay, so, now we're gonna look at how you do a function over this type. So this is the not function. It takes a Bool, produces a Bool. Bool is this new type we just added to the [inaudible]. So here's the function, so it takes a val one. If the val one is equal to true, so we're now using our type true then we are gonna return false, also we're gonna return true. Okay, pretty simple, these are just new things we have added to the language.

But what's even cooler is that you can use your types in a different way which is called pattern matching. The way pattern matching works is instead of doing before I can write the same exact thing and the same thing we just wrote as this. So this means the not function when given the pattern true will return false, when given the pattern false will return true. So I matched on the type that I want and I have a specific value for that type. Yeah.

Student: When you define a function do you always need to both give the typed information like the Bool arrow Bool and also the definition?

Guest Instructor (Sasha Rush):You don't actually have to do it. It will like figure it out for itself. It kind of uses documentation though, so it's like a good way of just telling other people that read your code, "This is what the function does itself."

Also, people write them sometimes because sometimes Haskell will figure out the type but it will be very generic so it will kind of generalize your functions for you and sometimes that can be a little bit confusing when you get error messages and things like that. So it's often good practice to kind of put that there.

Okay, so let's look at pattern matching a little bit more in depth. Here's the and function. This is kind of like the Boolean combinatory and. It takes a Bool, takes another Bool, and produces a Bool. So if one is true and one is false, it returns false. If one is false and one is false, it returns false. If one is false and one is true, it returns false. And if one is true and one is true, it returns false, okay.

Okay, so if you've programmed in C and newer versions of Java, you maybe familiar with this concept of an enum. An enum is often used in a similar way if you want to do true/false or colors or something like that where each one of these statements is assigned an integer value. So false maybe assigned the value zero and true maybe assigned the value one.

That's fine, but it's pretty much like a huge hack. Like it's really confusing because some things you can add different kinds of enums and things like that. So just to show you that these are not the same thing, so here is our apple type, here is our orange type. And if we try to take two values of these guys and compare them, it's going to fail. It's going to say, "Hey, these are not the same type." Just like when you tried to add a string to an integer, it's not allowed, that's a fail, we're done.

Okay, so now we're gonna talk a little more about user types. The other thing that is kind of cool about these types is that you can parameterize them with other values. Yeah, I'm sorry. So here's the orange type. So now we have this other orange and – well, so first of all, we have seeds and seeds has two values. You can be seeded or you can be seedless. So again, just similar to like true/false.

And then we have orange which looks the same as before except we have this other guy that has a kind of space and then another type put into it. So you can think of that guy as like an argument to the first type. So it's just when you use this in practice you can say, "Hey, I have my orange and my orange is a navel orange with seeds or it's a navel orange without seeds."

You can kind of put arbitrary other types as like arguments to the types originally. So this is kind of similar to a constructor in an object or a new language where you kind of make a new instance of the object you can kind of pass an argument that kind of are properties on the object.

Okay, so I said before that in Java any object can be just null [inaudible]. So I kind of discouraged that before because it can be really dangerous if you don't know an object is going to be null, it's kind of bad and you can cause failures. It also really useful in some cases because sometimes things just fail and you want a null coming back as kind of a marker as that hey, this guy doesn't exist, it can't exist, it was a failure.

So we can represent this guy in Haskell too. So here's what string, we are going to add a new type to our language called string stuff, and it can have two values. So the first value is that is can be null, and the other value is that it can have some stuff in it, and that some stuff pertains to the string that it's supposed to be, okay.

So it's just like a string in Java in that it can at any time be null or be some value and it has that valued contained within the type. So like this pretty good. It will work for strings, but let's say that I have some new data type like oranges and I want oranges also to possibly be a null or have some [inaudible] that we want to look at.

The problem is that this guy only works for strings, so I am going to have to write an orange stuff too. It looks pretty much exactly the same, but works for oranges. So what we really want to do is parameterize this definition here, not only with this value which is the argument base, but also with the type of that argument so that we can maintain our type safety without breaking things.

So is this kind of like lists in Java. So instead now we're gonna have a data type called stuff. So now I have this argument on the left side too which is like the type of the sum guy. So it's just like an argument to a function, but an argument to a type definition, and we can kind of carry that argument through. So now if I want a string I just type stuff string and then the some can contain a string element and so forth. Yeah. Student:

[Inaudible].

Guest Instructor (Sasha Rush):So you're talking about in this null?

Student: Yeah.

Guest Instructor (Sasha Rush):So the type of null would still be STR stuff. Let's see. So null since it's defined in the data as STR stuff, its type is STR stuff.

Student: So it will be reserved for STR stuff?

Guest Instructor (Sasha Rush):Yeah, sorry. You can't actually use it in both. Yeah, yeah, definitely, yeah, yeah. If I tried to type in both definitions it would just fail. And this works with any type A. Okay, so here's an example. So we're gonna try to do division, but we are going to make it safe so that if you try to divide by zero, we'd fail. So, if you divide by zero we return null. Oh, sorry, this shouldn't say object. It should say stuff.

So, when we try to divide by regular value, we return some, and then that value itself, okay. Why can't we just do this guy for the second part? So why do we have to have the word sum there? What's that?

Student:[Inaudible].

Guest Instructor (Sasha Rush):Yeah, well, so we definitely know the null one, but what would happen if we did this bottom guy? Why would it fail?

Student:[Inaudible].

Guest Instructor (Sasha Rush):Yeah, exactly. It would be returning an end type not like a stuff end type, which is the difference. Cool. Okay, so let's go onto to some more kind of cool types. So now let's do a list type. So a list type is similarly typed to what you have seen in Scheme. It has kind of a pair, like a cons to start with and then it has like an end value when you get to the end, and this is also abbreviated as we showed earlier in the talk as the colon operator.

So we say that a list is some value and then this list of that value, and when you get to the end you kind of have like a blank – like a stop type. This guy is a little bit crazier than what we have seen before because it's called a recursive type definition. You will notice that I am using the same definition that I used in the data side on the actual definition and that just lets us say like, "Hey, this value contains another value with inside of it."

So like a list when you do the tail operator contains another list right there. Okay, so here are some examples. This type string defined in Haskell is just a list of chars so everything is still consistent with this type definition we showed before. So, yeah, now why this is fail? Why do we have to have homogenous lists?

Student:[Inaudible].

Guest Instructor (Sasha Rush):Well, I know why we do it, but just because it's fast does not mean we should prevent people just because. So, let's look at this definition. So what was that type of that thing we just showed? So this guy is type list char because they're all characters. The guy at the top is a type list int because they are all integers. So when we try to do this guy we have integers and characters kind of intermingled in the type definition.

So we look back at this guy. Well, we said a list starts with a type, so if this guy is type int then we can only have type int inside of it, right? So we try to put strings inside of it or chars inside those two aren't going to match. That definition is gonna fail and the compiler is gonna throw an error. Okay, so there's nothing special here, nothing like magic keeping the list consistent, it's just based on what the type of the list is.

Student:[Inaudible].

Guest Instructor (Sasha Rush):So there are kind of like ways to do it, but you lose again like speed and safety. We'll show you one a little bit later that's one possible way of going about it, but it's kind of like [inaudible], how to do that well. Okay, so let's look at some kind of functions over these kind of objects.

Okay, so here like sum. So what sum does is it takes a list of integers and it returns a different integer. So again we're doing pattern matching just like we did before with true and false, but here the pattern is hey, is this a blank list. If it's a blank list, then we return zero. And then the other thing we're pattern matching on is just like we did before with pattern matching an int and a list of ints right in the function, okay.

We take those values, so we have a value next and a value rest, and we can kind of act on them just like they're values. So all we're doing here is adding next to the recursive sum of the rest of the list. Okay, here's the next guy. This is map. You have probably seen this definition before from Scheme.

So what is map? So this is the first time we have seen a function used as a type within another function. So as I said before, the way we read this guy is map is a function that takes a function from sum type A to sum type B. Then it takes a list A and it returns a list B, okay. So a map over an int list would take a function from ints to ints, take a list of ints and return a list of ints, okay?

So what do we do? Well, if a list is blank, we do nothing. If the list has some elements, we just apply the function to the first element, and then we put it at the start of another list that's the rest of the elements with that function. Yeah.

Student: Can you add [inaudible]? Would it compile?

Guest Instructor (Sasha Rush):Yeah, it would compile. You'd get – you actually would be totally fine in these cases because these two are different types. So think of this one like true and the second one like false, so you can have them in any order because it’s just matching is this type. Yeah.

I guess I should say one other thing which is that you can also write map fun and just give a variable so is they said map fun list or LS or something, then it wouldn’t work, so switch the two because the first one would match an empty list also. Does that make sense? So it matches the most general possible.

Okay, so this is my – let’s see if we can get some audience participation for this one. So here are some types that are a little bit more crazy. So if you got – this is like – I think this pretty interesting, if you can like look at them and just take a guess of what this actually is, like what are we trying to describe here? I think we have plenty of time, so we can probably go through these.

Okay, so it can be two things. The second thing is pretty boring. The second thing looks just like our sum from before, right? It’s just has one value in it, right? So, what’s going on with the first thing? The first guy is a little bit crazy.

You can have one value and then it can have [inaudible]. It’s a binary trick, right. So the first guy is like a branch and it has like two different paths that you can go down, and the second guy is like a leaf, which is just the value itself. Okay, so let’s look at this guy.

One of the things that’s kind of cool about type definitions is that a lot of times you can just look at them and say, “Oh, I know what that does just in the type definition.” That’s why I said before they are used as documentation. They are also how people like to search for new functions on the web. They’re like, “Hey, is there a function that does this? Let me type in a type definition.”

So what does this guy do? Well, it takes a function that takes an A and a B and produces a C. Then it takes a list of A’s and a list of B’s and produces a list of C’s. So the hint is we saw this guy earlier. Yeah, this is zip with. This just takes the function and applies it to the two lists one at a time, and produces the final list.

Okay, one more. So this last guy – yeah. So this guy looks very similar to a definition we saw earlier which is of a list itself, which is a pair and then a N, except that it’s got this craziness where it has two variables it’s taking to start with, okay. So we’re used to just seeing a list parameterized by one type.

This guy is parameterized by two types and you see this craziness where it flips the types when it goes to a [inaudible]. So this guy is like a list where the types alternate. So like the first one if you give it an int and a string, the first would be an int then a string and then a string, kind of alternates as it goes. Yeah.

Student: What is the capital A?

Guest Instructor (Sasha Rush):Capital A was just like me hiding what that would be. So in real practice you would give that a descriptive name like cons or something. Okay, so here's kind of just like how we would actually write these guys in practice. The first is a binary tree. The second is the function zip with. And the last is this kind of switch list guy, which isn't actually practical for anything, but is a good way to understand how lists with mixed elements work.

Okay, so you probably learned that [inaudible] to code is pretty good and we want to use it. So in Haskell there isn't really any object referring to stuff at all. Part of the reason of this is like functional programming [inaudible] because they think it's just like function code, so there's a whole history of those two movements.

The more practical reason is that it would be kind of hard to use it in Haskell. For instance, because you can't change any values, you don't have variables in the same way. It is kind of useless because there aren't any setters. You can't change an objects value. And because it's all garbage collected there is no such thing as a destructor, so all of that stuff doesn't exist. So some of the benefits I've been referring just aren't apparent, but there is some stuff that is really good. Let's kind of try to map these concepts back to what you are used to seeing.

So the main idea of a class in Haskell is this data type. So you make a data type for everything that you want, that you would make a class for, so those two are kind of similar. Then you can think of the actual use of the data type, like the use of true or the construction of a class of a list as kind of like being an object. So it's an object of that class and you can do stuff with it.

And then finally, we are going to look at this thing called type classes which is kind of poorly named but these things kind of act like interfaces in kind of a magic way. Okay, so here is an interface in Java. I forgot what a Bool looks like, but the idea is that anything that implements this interface must have a function called equals and a function called not equals. And then – sorry, a method called equals and not equals. And then we can always call those methods on any objects of that type.

So Haskell has a similar idea called a type class. So you make a class and you say, "Hey, this is a class called PQ and it applies to type A." And if this class exists that type A, there must exist these two functions, one function is an equals function that takes the type A, another type A and returns a Bool. And one is this not equals function which takes type A, type A, and returns type Bool.

The kind of cool thing that you can do in type classes that you can't do really do in interface is actually implement one in terms of the other. So the fault implementation of not equals is simply the not operator applied to equals. So now here's like an implementation of that.

So we say the instance of the equals class for the type Bool that we defined earlier means that if they're both true then that returns true, if they are both false, then that returns true, and if they're anything else then we return false, okay.

And what does this mean? Well, this means that anywhere in your program, anywhere you want, we can now use equals equals on Boolean guys and it will just work. It will just magically work. So here's another example that is a little more interesting.

So we defined this stuff class earlier and what we can do is we can say that hey, if you have a type that already has an equals, we've already defined the class equals for it like Boolean, and you have the stuff of that guy, then stuff is automatically defined with equals too by this definition. You can kind of do this for anything. You can say two trees are equal if this definition holds. You can say like two whatever's are equals.

Equal is like a pretty small example, but what's neat is there are these kind of general type classes that work over lots of things. For instance, a lot of objects in Haskell, the map operator will work for them too. So you can kind of imagine a map operator that works on trees that goes through each element, applies a function, and produces a new tree with those elements in it.

And things start to get really powerful when you can kind of just assign this arbitrary operator, have it work. You don't even have to know what is being passed in because you know as long as it defines map your function will work on it. Okay, so I think that's about it for my actual technical definitions. I said earlier though that you can't get a job with Haskell and that kind of made me sad because I know everyone wants a job.

So I thought it would be good to kind of like go through a couple of the interview questions that we give to people and show them how it would get done in Haskell. I found that people come in and they try to write interview stuff in C and they get so confused over like hey, did I forget this number here, or should this be less than or less than equal, and it's just pretty bad.

So I think one of the things that Haskell does is it treats you to think about the problem from kind of a mathematical definition and not get worried over the details of implementation. So how to get a job with Haskell.

Okay, so here is a common interview question. I figured I would ask a couple from Google and a couple from Facebook so that way we're equally screwed if you guys try to come in and interview. So this one is you are given two assorted numbers produced an assorted list of all the numbers, so it's kind of like merging the two guys together.

So the problem here is actually not really much of a problem at all. It becomes tricky when you do it in a language where you have to kind of create a new array that has the size of both of them, kind of like fill them in one at a time, and then return that array. In Haskell though, it's just a matter of taking both arrays, seeing which element of the two at the beginning is less, merging that into a new array and kind of continuing to go along.

So let’s look at the code. So the first thing, as I have said before, you always do is write down your definition of what’s going on here. So we have a list of integers, another list of integers, and we’re producing the merged list of integers between them. Okay, so here’s the code – if both lists are blank, we want to return blank. That makes sense, right? Nothing to do here.

If one list is blank and the other guy still has values, well, you definitely just want that guy because he still has values left, the other guy doesn’t, and vice versa. So that’s the first three lines. They’re just boilerplate. We’re just putting them in to do stuff.

Now the actual interesting part is the last case where we are actually doing the merge. So you will notice here that we can take the head of both lists, write in the function definition. We say if A is less than B, well, then we want to put A at the front of our list and the rest is just, hey, merge these other two lists that we created that we had already.

And then, hey, if B is less, then we take B, that guy starts the list, and then we take the rest of them and just merge them together. There’s not really any magic going on here. We’re not using any laziness. We’re not really even using our type inference. But it’s fast, it’s simple, and there we go. So there is variables where you call them whatever. It just means the tail of the list. So A is the head of the list, and AS is the tail of the list. Yeah.

Student: Do the interviewers care which language the interviewees chose?

Guest Instructor (Sasha Rush):Only if they’re jerks. Student:

[Inaudible].

Guest Instructor (Sasha Rush):Yeah.

Student: Is the less than sign the only thing in there that actually forces it to infer that they’re pink lists as opposed to just generic lists?

Guest Instructor (Sasha Rush):Yeah, and in fact, if we hadn’t have given this type definition in there, what would it infer?

Student: Just rely on probably using the less than.

Guest Instructor (Sasha Rush):Well, yeah, just using less than sign. So in actual practice we showed the EQ type class. We’d also have a comp type class. The comp type class would have less than, less than, greater than, all of those in there. So it would infer that the type would have to be something that has an instance of the comp class, so anything that would have that.

Okay, so here's another one, this is a function A2I. This one is a little bit trickier. So what is it? Well, it's a string, which as we said before, is just a list of charts. And we are mapping in to an int so it means convert an ASCII string to an integer, okay?

So here's the code. So again, four lines, they are not that long. The first thing we have to do is define this helper function which we are calling D to I, which means convert a digit to an integer. The common trick for doing this in C is simply to minus by the zero character. Does that make sense? Since ASCII characters are in order you can just minus by zero and get us back the digit.

Anyone know why we can't do that in Haskell? I mean characters are just represented by numbers, right, we should just be able to do that, right? Well, again as we said before, since there is type safety we want to distinguish the characters from numbers so in case you accidentally add them together by mistake, it's bad news.

So we have this function ord which tells us what the ASCII value is, and we apply that and do the subtraction. Okay, then this guy is really like the brunt of what we're doing. There's a little trick here called an accumulator. I don't know if you have seen that in Scheme. It's like the way to write recursive functions when you want it like kind of – it's kind of a more powerful way to write recursive functions.

So what are we actually doing? Well, for each character we are multiplying the previous stuff that we've seen by ten and we are just adding on what that digit conversion was. So we go through, so if our numbers are one, two, three we first get the digit one. Then we multiply that by ten and we get digit two, and we add that on multiply it by ten, get digit three, add it on, and then we're good.

Okay, we're running out of time. I have one more example, and then I guess you want to finish up. Okay, so, here's a function to check if a string is like a prefix of another string. So the type definition is list of chars, list of chars to a Bool.

Okay, so what are we gonna do? Well, we are assuming that our first guy is the prefix and the second guy is the string we are testing again. So each time we just pull off the first character. If the character is the same, if they're equal, then we keep on going. We do the rest of the list. If they're not equal, we've failed because it's definitely not a prefix so we return false.

And then we just handle the two other cases. The cases are, hey, your prefix is blank. Well, that means we got to the end, so if we got to the end then it's definitely a prefix, so that's true. The other case is that the string we're testing against is false is blank, and that means the prefix was longer than the original string, so bad news, so that's false.

So that one's three lines. It's pretty simple. Cool, so anyway, I hope that this at least opened your mind to there being really interesting other languages out there. Haskell is pretty cool because it's both a research language and it's becoming more and more a practical language. There’s a really great wiki, Haskell.org. It will teach you all kinds of other crazy magic in Haskell.

The particularly big one is I haven’t showed you how to print yet, and as I hinted at earlier, it’s kind of difficult to print when it’s lazy, so there’s are kind of really cool tricks for that. There are a bunch of different implementations. GHC is the main one. You can download that for your Mac or whatever. And there’s a bunch of big programs being developed in Haskell like right now. So anyways, thanks for having me.

Instructor (Jerry Cain):That is all. I will see you all Monday morning. Don’t freak about the exam. I’m not saying it’s going to be easy, but it won’t kill you either. Okay, so I’ll see you all then. Get back to you midweek with your finals. Bye bye.

[End of Audio]

Duration: 58 minutes