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