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