ProgrammingAbstractions-Lecture26

Instructor (Julie Zelenski):Hi, there. It's Wednesday. We are coming to the end of this fine quarter. So let me tell you a little bit about administrative stuff that's from here to the end so we're keeping track of all these things. Then we'll go on and talk about what we're going to do today.

Today, I'm leaving to go out of town to go to a conference in Portland, a special interest group on computer science education. So all the CS teachers in the country get together and talk about how to torture their students for a couple days in Portland. I will be leaving at 4:00 p.m. to head straight to the airport. So if you need to see me, and you're hoping to come to today's office hours, please come at the beginning there so I can take care of what you need before you leave.

I will be out of town, but hopefully email accessible. If you needed to talk to somebody in person, Jason is the guy to follow up with if you need to see something before I come back at the end of the weekend.

We will meet on Friday and have an optional guest lecture, but I think it's actually very valuable. It's Keith Schwartz, who is the instructor for the cs106l lab class. So those of you in the lab class already know what a superstar he is. Those of you who haven't been in that class will get to see what he's all about. He's going to come and talk to you about C++ without 106. So we teach a certain variety of C++. We have a subset of language we pick. We have a set of libraries and tools, and we use certain features of that.

Our pedagogical goals, but there's a lot more to C++. So Keith is going to give you an overview on Friday. It's kind of the whirlwind tour of if you were out of 106 and you were staring to use C++ on some project you're doing or some further classes or some internship you have, what things do you need to pick up to complete the picture of how the more professional standard C++ looks as distinguished from 106b's version.

So I think that's a very useful and very practical thing to take away from this. So if you actually are available to join Friday, I think you'll get a lot out of that. Then other things that are happening, Pathfinder's coming in this Friday, so the last bit of heavy travail you have to do for us is get that assignment in by this Friday. There are no paper copies do, so all e-submit. We won't be doing the interactive grading in that sort of form. All we need is your electronic submission to get our grading done on that.

Our exam is a week from this Friday, so the very end of the exam week, we will be meeting in Krezgi auditorium, which according to the registrar's database, seats 492 students, which is enough for all of us plus some. That's over in the law school. That is the place for 106b fun to happen next week during exam week.

I think administratively [inaudible]. So what I'm going to do today is I brought a cup filled with candy. We'll start with that because that's going to be helpful. What I'd like to do is it's just a couple bits of loose ends, some things that I wanted to touch on that have

come and gone this quarter. I maybe wanted to pull them together and reach a bit of closure on. Then hopefully, if we have time, we'll move on to what happens after this. If you liked 106b, what do you do? If you don't like 106b, what do you do? What kind of options and future opportunities are there after this ends? I'm going to talk a little about some of the things that [inaudible], but I do want to talk about the final just for a second because it is kind of a last opportunity for you to show us your mastery of the material and get the grade that you've been working hard for all quarter. It does carry a certain amount of weight. 30 percent of the weight in the default system is actually placed on the final, and more of it can move to that if you have a better showing on the final than you did in the midterm. We're happy to move some of that weight over so it could actually account for even more of your total grade.

So there's a lot riding on it, and you want to be sure you come ready to show us the things that you know and that you have comprehension and mastery over. To note, though, it is going to look a little bit different than the midterm in terms of its emphasis. We are going to see a lot more of this nitty gritty dynamic data structure, Linklis, trees, graphs, pointer-based structures with the inherent complications in dealing with that kind of dynamic structure. So bringing your best game to this exam is certainly going to pay off here.

The coding that you will see will look a lot like what's in the assignments. If you already had a chance to look at the practice exam, you'll see that there are themes there that echo the things you did on the work. You want to try to make sure that's we're connecting to the effort you've put in all along. There will actually be some opportunity to think a little bit at a higher level about making and analyzing choices for design and implementation. I'm going to talk a little bit about that today because I feel like at any given moment, we're talking about how we might implement a stack and how we might implement a queue. Sometimes it helps to step back a level and think a little bit about the choices for the bigger picture of when to use a stack and queue and how to make decisions in the large about those tradeoffs that are intelligent and grounded. So I'll do a little bit of that today, and then some of that has been built up all quarter long.

It is open book and open notes. You can bring all your printouts, your readers, sections, all those sort of things. For some people, that translates to this idea that, well, I don't need to prepare because I'll have all my stuff there. If I need to know something in the exam, I'll just go look it up. That's certainly true for looking up details. You want to remember how this works or you had a piece of code that you think was similar. It could help you to brainstorm what you're trying to do this time. But you're not likely to have time to learn new things n the exam. So if there's something that you really feel you have not gotten your head around, the investment upfront before you get into the exam is where to get that understanding in place rather than trying, in the exam, to be flipping through chapter ten, trying to learn something. It may not really play out.

I highly recommend practicing on paper in an exam-like environment. So really working the way you will be working in the exam to give yourself the best, consistent prep for what's happening there. Then some people have asked about extra problems, just wanting more of them to work on. I'll give you a couple places where you can look for things.

One is that cs106 is being offered this same quarter, and they are just the honors version of our class. So they have some slightly notched up problems. But their practice and real midterm are posted on their website as well as their practice final. They have very similar coverage. They did some assignments that were like ours, some that were a little bit different, but still go places to kind of just grab actual exam problems with their solution.

The chapter exercises and section problems often are old exam problems or eventually became exam problems. So they kind of come and go in terms of being used as examples or used as exam problems. So they're definitely places to pick up extra mileage.

Okay. So let me tell you a little bit about – any questions about the final?

Okay. Let me talk about this line just for a second because I feel like we have touched on these themes again and again, but maybe it's good to have at least one moment of trying to pull this together and think a little bit about the closure and the big picture of all these things. A lot of what we're talking about in the beginning was we were giving you these abstractions, these cool things to do stuff with, a map, a stack, a queue, a vector, and problems to solve where those abstractions have a role to play. Then we spent a long time saying, okay, now that it's our job to make those abstractions work, what techniques can we use? What algorithms and data structures will help us manipulate and model those structures in efficient ways.

We certainly spent a lot of time talking about runtime performance as though that was maybe a little too much in your mind to think that was the only thing that really mattered. How fast can we make something? O of N is better than N squared. Log N is better than N, driving to these holy grails of bringing the time down.

It's certainly a very important factor that often – the difference between something being linear and something being quadratic is a matter of it being tractable at all for the problem at hand. You cannot sort a million numbers in a quadratic sort in nearly the same time. For large enough data sets, it'd be completely unfeasible. So it is important to have that big picture, to be thinking about it and know about those tradeoffs, to understand issues of scale here. But we also did some stuff on the homework about actual empirical time results, which is another way to bolster our understanding. The big O tells us these things, but what really happens in real life matches it, but not precisely. Those constant factors, we threw away, and other artifacts that are happening in the real world often give us new insights about things and challenges that we're facing.

We also need to think about things like mix of expected operations, having some be slow at the consequence of other operations being fast. It's often a tradeoff we're willing to make. On the editor rougher, we drove toward getting all the operations to be fast. In an ideal world, that's certainly the best place to be, but it's also a matter at what cost in terms of code complexity and memory used and effort put into it. It may be that we're willing to tolerate some operations being less efficient as long as the most commonly used ones are very streamlined.

We talked about these worst cases and degenerates about knowing when we can expect our algorithms to degrade and whether we want to do something about that or choose a different algorithm or provide some protection against those kind of cases coming through. To a lesser extent, we already talked about memory used, how much overhead there is. Seeing that we went to a pointer-based Linklis structure, we added the four byte overhead for every element. We went to an eight byte overhead once we got minor [inaudible] and maybe even a little bit more if we had balance factors included.

Those things ride along with our data, increasing the size and the memory footprint of it. So there's a certain amount of overhead built into this system. There's also this notion of excess capacity. To what extent do we over allocate our data structures early, planning for future growth. In some senses, save ourselves that trouble of enlarging on demand. It's preparing for a large allotment and then using it up when we find it. Then when we do exceed that capacity, having you do it again.

So where do we make those tradeoffs about how much excess capacity and when to do the enlarging? It has a lot to do with what our general memory constraints are. I would say that memory has gotten to be quite free in recent processors and computer systems. You don't really think about 100 bytes here, 200 bytes there, 1,000 bytes there. Even megabytes, you can talk about as thought it's a drop in the bucket. So this is something that isn't given as much weight nowadays as it was ten years ago when there were a lot more constraints. But with the move for a lot more imbedded device programming, and looking the iPhone or cell phones or things like that where they don't have that same liberty to be wanton about memory, it's come back to being a little more careful and keeping an eye on it.

There's also and issue here which trades off a lot with runtime, which is redundancy versus sharing. Having two different ways to get at the same piece of data often will provide you with quick access. For example, if you have a card catalogue for a library where you can look up by author, you can look up by title, you probably are maintaining two data structures. One that manipulates them, sorted by title, one that manipulates them sorted by author. They're both kind of looking at the same books in the end. So maybe there's this set of pointers in one map over here, indexed by author, another keyed by title over there.

That means that I'm repeating all my data. Hopefully I'm sharing it with a pointer, so I'm not actually really repeating all the data, but I actually have two pointers or potentially two or more pointers to the same book, depending on the different ways you can get to it. That gives me that fast access. I want to know all the books written by Leo Lionni, I can find them easily, as well as finding all the books that have King in the title. I don't have to do searches on a data set that's been optimized for only one access. Definitely, tradeoffs were more space thrown at it to get at that fast access for different ways.

Then this last one is one that I had mentioned along the way, but I think it's one that's easy to overlook, which is to get very excited about how fast you can make it, how small you can make it, how fancy it is. It has a real downside in terms of how hard it was to

write. Typically, when you go to these fancier strategies that are space efficient and time efficient, you had to pay for it somewhere. It wasn't for free that the easy, obvious code would do that. It tends to actually be complicated code that may use bid operations, that probably uses more pointers that has kind of a density to it that means it's going to take you longer to get it right. It's going to take you more time to debug it. It's going to be more likely to have lurking bugs. It's going to be harder to maintain.

If somebody comes along and wants to add a new operation, they open up your code, and they're frightened away. It might be that that code will never get moved forward. Everybody will just open it and close it immediately and say, whatever. We'll make it work the way it is, rather than trying to improve it or move it forward because it actually is kind of scary to look at.

So thinking about what's the simplest thing that could work. There's this movement that I think is pretty neat to think about. There's this idea of the agile programming methodology. It's based on the idea that rather than planning for building the most super stellar infrastructure for solving all the worlds problems. You're about to write a chess program. Actually, something that's even simpler. You're going to write a checkers program. You say, I'm going to sit down, and I'm going to design the pieces and the board. Then you think, hey, checkers is just one embodiment of this. Why don't I try to make the uber board that can be used for strategic conquest or for chess or Chinese checkers. You could have these hexagonal things.

You start building this crazy infrastructure that could handle all of these cases equally well, even though all you really needed was checkers, a very simple, 2D grid. But you imagine the future needs way in advance of having them, and then you design something overly complicated. Then what happens is you get bogged out of that. It never happens, and the project dies. You lead a lonely life by yourself, eating oatmeal. I like oatmeal.

So one of the mottos is design the simplest thing that could possibly work. I think that's a neat gift to yourself to realize that this simplest thing that could possibly work, that meets the needs that you have, that has a decent enough [inaudible] and memory constraint right there. Meets your needs in the simplest form of that. It's going to be the easiest thing to get working and running. If you later decide you want to go fancy, you can swap it out using the principles of distraction and [inaudible]. It should be that it's independent when you decide you need that fancier thing.

I've got a couple questions here because I thought this would just help us frame a little bit of thinking about these things. These are questions that I'm hoping you can answer for me. That's why I brought cup full of candy. I'm prepared to offer bribery for people who help me answer these questions. So let's talk about array versus vector. So the sequels [inaudible] builds and array is what is behind the scenes of a vector. Then the vector adds convenience on the outside of it.

Early on, I had said, once I tell you about arrays, put that back in your memory, and just use vector instead because everything array does, vector does nicely and cleanly and adds

all sorts of features. Somebody give me a list of things that vector does much better than array.

Way in the back.

Student: Bounce checking. Instructor:

Bounce checking. What else? Come on. A blow pop is on the line. I need to interrupt your – help us out. You've got bounce checking. What else?

Student: More? Instructor:

I want more. You've got nothing else for me?

Student:I don't even want a blow pop. Instructor:

You don't want a blow pop? All right. I'm going to give your blow pop to somebody else. Right here. I'm going to give the blow pop to you. Tell me what makes vector good.

Student: It has operations that shuffle elements for you. Instructor:

Yeah, it does the insert, delete, moving stuff down. Also, it does the growing. You keep adding, and it just grows. [Inaudible] don't do that. You want a [inaudible] to grow, you grow it. You take the pointer, you copy stuff over, you delete the old one, you rearrange things. You do all the work.

So vector buys you kind of array with benefits. So it seems like, then, you could kind of take away – the naïve point would be like, well, just use vector. Always use vector. Vector's always better than array. Is there ever a situation where array still is a pretty valid choice? What is it array has that vector maybe doesn't?

Student: Well, when you add things dynamically to the vector, it actually doubles the size [inaudible] increasing it by one. If you know the exact size, and you know it's not going to change, you're using [inaudible]. Instructor:

That's it exactly. One of its big advantages. If you know you only have – let's say you have a very small array planned for. The one on the practice exam was, if you're doing a calendar where you had 365 days in a year, and you're keeping track of the events on a particular day, you know how many days in the year. There's never going to be any change in that. Well, leap year.

But you know that, and you know that in advance. So why fuss with a data structure that does all this growing and shrinking and has some overhead associated with that when, in fact, you know there's exactly this number. That's all you need. In fact, it's almost a little bit more convenient to set up. You can just declare it that size, and it's ready to go. With the vector, you have to go through and add all the elements and stuff like that.

So there are a few situations where you might just say, I don't need the advantages. It adds some overhead that I don't want to pay, and I'm perfectly happy making it a small fixed array. You'll still lose bounce checking, but if you are being careful, maybe you're comfortable with making that tradeoff. By losing bounce checking, you actually are getting a little bit of the speed improvement back. The bounce check does actually cost you a little bit on every vector access that an array would not actually pay.

That's really in the noise for most programs. I hesitate to even mention it, to get you to think it matters, but it is part of the thinking of a professional program. Sometimes that may be the thing it comes down to, making your operation constant factors streamline enough for the operation you need.

So I put here stack and queue versus vector. Given that stack and queue are really just vectors in disguise, and they're vectors with less features, vectors that can't do as much. One of the implementations for stack and vector could easily be just layered on top of a vector. Why is it good to take away things?

Student:[Inaudible] tampering with contents of [inaudible]. Instructor:

Yeah. So it's enforcing discipline on how you use it. Stacks are lifo, queues are fifo. The way things go out, they way they go out are very rigid. By using stack and queue, you are guaranteeing that you won't go in and accidentally put something at the head of the line or remove something out of the middle of the stack because you just don't have access to that. It's been tidied up and packaged up into stack, which has push and pop. queue, which has MQDQ.

No other access implied or given, totally encapsulated from you. So in fact, it allows you to maintain on a discipline that otherwise the vector didn't strongly provide for you. Question?

Student:[Inaudible]. Another thing you could do is you could optimize a stack or a queue to be very good at the only active things you're doing, which is pushing and popping or RQDQ. Instructor:

I think that's worth a smartie, don't you? I think that is. Exactly. So by restricting what you have available, it actually gives you choices in the implementation that wouldn't have worked out as well, [inaudible] full array of operations. It's exactly true. Stack and queue only adding from one end mean that things like a Linklis, which isn't a good choice for implementing the general case of vector, are perfectly good choices for the stack and queue case.

So giving ourselves some freedom as the implementer. So information about how it's being used can pay off. It also means when you read code – if you see code that's using something, call the stack or using a queue, you immediately know how it works. You could say, here's this search that's actually stuffing a bunch of things into a stack. I know they're last and first [inaudible]. If you see them using a vector, you have to verify where

they put them in the vector. Where do they pull them out? The code doesn't tell you the same things. It doesn't tell you the same story as it does when you see a word that has a strong meaning to it, like stack and queue.

Generally, these [inaudible] to make about sorting things or not sorting things. We spent a long time talking about sorting and how various algorithms approach the sorting problem and what they come up with. I have a question here. What sorting algorithm is best? Is there an answer to that question? Why would I ask a question that has no answer?

Student: It depends on what you expect it to be giving you. Instructor:

It depends. That's a great thing. It depends on what you expect it to give you. How big is it? What are the contents of it? Is it random? Is it sorted? Is it almost sorted? Is it reverse sorted? Is it drawn from a range of one to 1,000 values, or is it drawn from one to three values? Huge array that has just one, two, three in it might necessitate a different approach than one that has one to max sine in it.

So there is no answer to that. It is depends. What do you know? If you don't know anything, give it to someone else. If you don't know, there's certain things that perform well. You know in the average case, you don't have degenerate behaviors. For example, [inaudible] is a great sort for the general case of an N log N that doesn't use any space or any extra space the way word sort does. It doesn't have any degenerate cases. For an unknown set of inputs, it works pretty well.

If you happen to have a little bit of data, it's almost sorted. Even Barack Obama's not favorite bubble sort has a chance of being in the run because it's already sorted.

Is it worth sorting? Calculating the payoff, we had a problem on one of the section exercises once. I think it's a really good one to think through and revisit. This idea of certain operations, many operations are much more efficiently implemented if the data's sorted. You can think of them off the top of your head. Oh, you want to have the minimum? If it's sorted, it's very easy. You want the maximum? Very easy if it's sorted. You want to find the median. Also very easy if it's sorted. You want to find duplicates. Once they're sorted, they're all lined up together. You want to find the mode, which is the most frequent element. Also easier if it's sorted because they'll all be groups together in a run. So you can do a linear pass, counting what you see to find the longest run.

If you had two arrays and you wanted to merge them or intersect them to find their common elements, it's much easier if they're sorted. All these things that are hard to do when the data's in a random order actually have much more streamlined algorithms once you invest in sorting it. So there was a problem that was like, how do you know it's worth doing that? If you only plan on writing one merge, and you could do it in N squared, is it worth it to sort it, N log N, so you can get an O out of merge?

So it has a lot to do with how often you plan on doing the merge. How big is your data set? What are your constraints on the times there. It is interesting to think about that

relationship. It isn't for free, but potentially, it's an investment by sorting it so you can do a lot of fast searches later.

This one actually is interesting because they actually solve a lot of the same problems, and they differ in one small way. If you had a set – so set is being backed by a balance by a [inaudible] string that actually is a sorted data structure internally – versus a sorted vector. So if your goal were to keep track of all the students at Stanford, and you wanted to be able to look up people by name, then the contains operation of the set is actually doing these minor ray search on assorted vectors using the same path, but working through a vector.

So we can make the searching operations, like contains and other look-up based things, algorithmic in both cases. So what advantage – this sort of vector, where I just used the interchangeably, is there something one can do that the other can't or some advantage or disadvantage that's assumed in that decision?

Student: You don't have duplicates in a set, but you can have duplicates in your sorted vector. Instructor:

Yeah, so the set has another concern with just how it operates. You can't put duplicates in, so if you have two students with the same name, then in the set, it turns out you're all coalesced down to one. Your transcripts all get merged to your advantage or not.

What else does set buy you that vector doesn't? How hard is it to edit a sorted vector? You want to put something new in there? You're shuffling. You want to take something out? You're shuffling. The movement to the pointer-based structure there, the research stream behind this, is giving us this editability of the data structure. The convenient logarithmic time to remove and add elements using the pointer wiring to do that, that vector doesn't have.

That tradeoff is actually one that situation may be that you just don't do a lot of edits. That's actually a very common situation. The sorted vector happens to be a very underutilized or under appreciated arrangement for data because for most things that are getting a lot of edits, having a linear time edit isn't going to be a painful consequence. You're doing a lot of searches, and you're getting the finer research there, where you really care about it, with no overhead. Very little overhead. A little bit of excess capacity, but none of the pointer-based trouble that came with the set.

So the set actually adding onto that another eight bytes, potentially even a little bit more, of overhead on every element to give us that editability that maybe isn't that important to us. So if you're looking at that thing and saying, I'd like to be able to have a sorted structure for both the set and the sorted vector, let you access the elements in order. The interrater of the set goes in sort order, going from zero to the end. The vector gives me order. I can browse them in sort order. I can search them using sorted. Where they differ is where does it take to edit one? The one that invested more in the memory had faster editing.

The P-queue. This is kind of interesting. The P-queue is also a sorted structure, but not quite. It's a much more weakly sorted structure. So if you're using the [inaudible] heap like we did on the P-queue [inaudible], it does give you this access to the maximum value, and then kind of a quick reshuffling [inaudible] the next and so on. But it doesn't really give you the full range. For example, if you were looking for the minimum in a P-queue, where is it?

It's somewhere in the bottom. It's in the bottom-most level, but where across the bottom, no knowledge, right? It could be at any of the nodes that are at the bottom of the heap. So in fact it gives you this access to a partial ordering of the data. In this case, it's relative to the prioritization, but not the fluid form of sorting the way a vector is. For example, if P-queue doesn't give you things for finding duplicates or finding the minimum or doing the mode it's not actually an easy operation because you have it in a heap form. I can get to the max quickly.

So if I cared about pulling them out in sorted order to browse them, I could stuff things into a P-queue and then pull them out, but what's interesting about that is it requires destroying the P-queue. The P-queue isn't really a place you store things and plan on using them again. It's the place you stick things in order to get them out in an order that's only value is in the de-queuing of them, pulling them out.

So finding the most promising option to look at in a search or finding the most high-priority activity to take on or something. But it doesn't really have the – if I wanted to be able to look at all the students in my class in sorted order, it's like, well, I could stick them in a P-queue and then pull them out, but it's sort of funny. I have to put them in the P-queue just to pull them back out. It wasn't a good way to store them for that browsing operation to be easily repeated. If I put them into a sorted vector, I could just keep iterating over it whenever I need to see it again.

So a lot of the things that it comes down to is having a pretty good visualization of what it is that going to a pointer-based data structure buys you, and what it costs you, relative to the continuous memory. That's one of the tensions underneath most of these things. It's like, putting more memory into it by virtue of having a pointer wire as opposed to a location arrangement. There's no information that's stored to get you from a race of zero to a race of two. They're just continuous in memory. So it saves you space because you only have to know where it starts and access continuously for that.

This pointer means there's more kerversals. There's more memory. There's more opportunity to air, but it gave you this flexibility. It broke down this continuous block, that's kind of a monolith, into these pieces that are more flexible in how you can rearrange them. So thinking about these things is an important skill to come away from 106b with that, yeah, there's never one right answer. You're investing to solve this other problem, and there's going to be downsides. There's no obvious triumph over all. It's fast, it's cheap and it's small and easy to write solutions out there.

So I put up here – I had to whittle this down to the five or six things that, at the end of 106b, I want you to feel like, that's what I got. That's the heart of the goal. The MVPs – the most valuable players of the 106b curriculum here. You're looking back at it, and you go, where did I start, where did I end up, and how did my knowledge grow? I'm hoping these are the things that stand out for you. That was a real growth area for me.

One is this idea of abstraction. That was stressed early on. Here's a stack, here's a queue, here's a set. They do things for you. We don't know how they work yet, but you make cool things happen with them. So you solve maze problems and random writers and do neat things with dictionaries without knowing how they work. So you're leveraging existing material without knowledge of how it works. It helps to keep the complexity down. You can solve much more interesting problems than you could without them.

Try to go back and imagine the Markov random writer problem. Now imagine doing it without the map in play. So you had to figure out that given this character, go look it up and find a place to store it, and then store the character in some other array that was growing on demand, as you put stuff in there. You would never have completed that program. You would still be writing it today, having pointer errors and whatnot behind the scenes.

Having those tools already there, it's like having the microwave and the food processor, and you're about to cook instead of having a dull knife and doing all your work with a dull knife and a fire. You actually have some real power there.

Recursion's a big theme in the weeks that followed that. How to solve these problems that have this similar structure, looking at these exhaustive traversals and choice problems that can be solved with backtracking. Having this idea of how to think about thing [inaudible] using the recursive problem solving and how to model that process using the computer was a very important thing that comes back.

We look at algorithm analysis, this idea of making this sloppy map that computer scientists are fond of, talking about the scalability and growth, knowing about curves and the relationships and how to look at algorithms. The more formal analysis. Still, in this case, there's still quite a bit more formulas to this than we really went into. If you were go on in CS and look at it, you will see there's quite a lot of rigor that can be looked at in that area.

We were getting more an intuition for how to gauge things about algorithms and their growth. Then this backside, all about implementation strategies and tradeoffs. So now that we've benefited from those abstractions, and we've solved some cool problems, what's it like to actually be the person building them? What are the techniques? What does it take to wrestle a pointer or a Linklis or a tree, heap graph into doing something really cool for you. How does that inform your information, what you expect about it and the coding complexity that was inherent in achieving that result?

Hopefully in continuing with the theme that 106a started you on was this appreciation for [inaudible]. In the end, getting code that works, that's ugly, just isn't, hopefully, what you're satisfied with. You want to approach problem solving to produce beautiful, elegant, unified, clean, readable code that you would be happy to show to other people and be proud of and maintain and live with. The kind of work that other people you work with would appreciate being involved with.

Things that are not so much – I don't think these are the most valuable pointers, but what came along the way, the idea of pointers and C++ being part of the – C++ and its pointers, they're one and the same. We choose [inaudible] for a language, so as a result, we get to learn some C++ because it's so heavily invested with pointers. We also have to do some practice with the pointers to get our jobs done. I think of those as being off to the side. I don't think of those being the platform of things I worship. They are things we need to get our job done.

I did put a little note about pointers here because there were quite a lot of comments on the mid-quarter evals that said, pointers, pointers still scare me. They make me crazy. I don't know enough about pointers. Here's a fact for you. You'll probably never feel like you know enough about pointers. Today, tomorrow, ten years from now, you'll still kind of feel like there's some real subtlety and trickiness in there.

They're an extremely powerful facility that gives you that direct control over allocation and deallocation and all the wiring and sharing that is important to building these complicated data structures. But there are so many pitfalls. I'm sure you've run into most of these, if not all of them at some point, where you mishandled a pointer, failed the allocator, deallocate, and the way it gets reported, whether the compiler or the runtime error that you see is helpful in sorting it out is a real cause for long hours to disappear in the name of mastery of this.

If you're still wary of pointers, you probably should be. Think of this as your first go-around. We're still at the journeyman's stage for programming here. So although there were pointers in 106a, they were very hidden. This is the first time you really see it and are exposed to it and trying to get your head around it. If you go on to 107 and 108, you'll just get more and more practice. It will become more solid as you keep working your way through it, but at this stage, it's okay and commonplace to feel still, a little bit – you see the star, and you take a double take. That's just where you should be.

It's just like in the Chinese restaurants. They put the little red star by food you want to be careful of. That's why they put the star for the pointer.

I also wanted to zoom out farther, a decade from now, when you are thinking back on the whole of your education and what you learned. Do I want to really remember how to type in keyword and the sequel plus generic template facility? No. If you code in C++ every day, you'll know that. If you don't, you can forget it. It's long gone.

What I hope you'll carry away from you is a couple concepts that broaden outside of CS. One is this idea of algorithmic thinking, how you approach something logically and step-wise and draw pictures and analyze it to understand the cases and to debug when things aren't working well. I think that's a skill that transcends the art of computer programming into other domains, any kind of thinking logically and problem solving and analyzing in that way.

I also hope that some of the experimental things we did with the sort lab and the P-queue and some of the big O and thing we tried to do in class together help you to [inaudible] back of the envelope calculations. This is something I'm a little bit of a – it's a pet issue of mine, which is, I think, in the era of computers and whatnot, really easy to get distanced from numbers and start thinking, I just punch numbers into formulas and one comes out. It must be the truth. Not having any intuition as to whether that's the right number, whether that number makes sense, whether it's grounded, and whether you actually are [inaudible] or so off because you've actually made an entry error, and you don't even notice it.

So becoming comfortable with this idea of using numbers to drive things, being able to take some time trials, do some predictions, do some more time trials, match those things up, do a little sketching on your numbers and see the things that are working out. So being comfortable having the math and the theory reinforce each other and not get too distanced from the fact that those numbers do play out in the real world in ways that are interesting. Don't just let the numbers themselves tell the story. There really is more to connect it up with.

This idea of tradeoffs. You may not remember how you can write a Linklis or how the hash table does the things it does and how to get one working, but hopefully you will take away this idea that the answer to most questions begins with the phrase, it depends. If I say, is it better to use a hash table or [inaudible] search tree, the answer will be, it depends. There's no, it's always better to use this sort. It's always wrong to use this approach. It's, it depends. Do you have the code written for that, and it works well, and you don't care how long it takes? Then bubble sort actually could be the right answer. Barack Obama not withstanding.

But knowing whether memory is at premium, time is at premium, what your mix of operations are, what language you're working with, what program expertise you have, what timeline you have for developing it all can play a role in deciding what choices to make and what strategy to pursue. So being very comfortable with this idea that it's about gathering information before you can commit to one strategy as opposed to it's always going to be this or that.

So that's the 106b thing. I have a couple slides here on things that happen after 106b. I'm just going to go ahead and show you. This would be a great time to ask questions. In particular, if there's anything you're curious about. Where do we go from here, and how do we make good decisions about things that follow?

The most obvious following courses from CS – if you took 106b and you hated it, I'm very sorry. Hopefully the scarring will heal over and you'll lead a productive and happy life elsewhere. If you did like it and you think, I'm kind of interested in this. What do I do next to explore further? Where else can I go with this? I put up the intro courses with their numbers and names, and I'll talk a little bit about what each of these does to give you an idea of what opportunity it offers to you.

The obvious follow onto the programming side – 106a and b are programming courses, building your programming mastery. 107 follow in the same vein. We typically call those systems courses in CS nomenclature. That means practical, hands on programming, getting more exposure to the system.

So 107 builds on 106b's knowledge in a class called programming paradigms. Part of what it's trying to explore is different languages. So you actually look at the language, C, kind of stepping back to old school. You look at the language scheme, which is a functional language that was developed for AI purposes that has a long history coming from the math side of things. Probably looking at some scripting and flexible [inaudible] like Python. We kind of mix them up a little bit, so it's not exactly the same things, but looking at other languages, other ways of doing things.

The other component of 107 is also looking at some more low-level things. What happens? When you pass your code off to the compiler, what is it doing? What kind of analysis does do of your code? How does it actually generate machine code? What happens in the machine layer, understanding some more of the performance implications and how things are expressed. Getting a better understanding of pointers actually comes up because there is a certain amount of low-level coding that's being explored in this class.

So building on the programming skill as we move toward a Unix Linux-based environment. So away from the Mac and Windows. Just continue building larger programs with complicated features to grow your mastery. Internally, we call this class programming maturity, even though it's not its title. We see it that way in the sequence in terms of just growing you into a more versatile programmer, seeing different languages, different things and getting more understanding of what happens under the hood.

It is a five-unit class. It's typically offered fall and spring. Most people thing it is about as much work as 106b. Some a little more or a little less in different parts of the quarter, depending on what your background is and how 106b worked for you. In a model, I think most people think of it as being – would you guys say 107 about as hard as 106b?

Some ways, it's harder. Some ways, it's not. Maybe that's the deal. The 108 class which follows onto 107, but that prerequisite is particularly strong. It's a Java class. So moving in the other direction. Rather than moving down an extraction, moving up on the extraction, looking at more large-scale design, modern programming technique, using object-oriented facilities, thinking about large-scale programming. You do a big team

project in the end, so a three week, three person, massive effort that is the first really big scale thing that you'll be exposed to in the lower-division curriculum.

Student:[Inaudible]. Instructor:

Sometimes in 107l, which has some relationship to the 106l in the way that it's more hands-on, more C++, I don't know whether it will be offered this spring or not, but if you look in Access, it will tell you at least whether we have it tentatively on the schedule. I think that comes and goes with the interest from the students and the availability of someone to teach it. Student:

[Inaudible] can we watch [inaudible]? Instructor:

Yeah, 107 and 108 are offered on TV usually once a year. 107 is offered on TV this spring. The one in fall is not taped. It might be that the old lectures from spring are somewhere that you could dig them out, but I actually don't happen to know for sure. 108 is on TV, I think, in the fall, not in the winter. So the fall lectures are probably still around for 108, so you could take a look at them.

You could certainly look at their websites, and there's often a lot of old materials and handouts and stuff that gives you at least some idea of things that have been covered in the most recent offering of it. Certainly showing up in the first week is one way to get an idea of what you're in for.

108 is a four-unit class. Because it's in Java, there's certain things about Java, the safety and the attentiveness of the compiler, makes the error handing a little bit easier. So I don't think most people think of 108 as quite as intense as 107 or 106b, but that project at the end has quite a reputation as being a real time suck. So it has lots to do with being scheduled and motivated and working through that team project at the end. I think that's where most people find the big challenge of 108.

Then on the math side, in the nomenclature for CS, we have this idea of theory, introducing you to more the formalism for big O and discrete structures and doing proofs and thinking about logic. So we have a 103 a and b and a 103x, which is a combined, accelerated form of a and b, which serves as math classes. They are math classes for introducing the kind of mathematics that are important for computer scientists. So that is for people who are thinking about a CS major, another good course to get in early. Pretty much everything in the upper division of the CS major has these things layered down as prereqs. So these are the intro courses that serve as the gateway to prepping you to go on, and then looking at things in a more topical way.

Looking at networks or security, HCI or artificial intelligence layers on a grounding in theory and programming that came from the intro courses.

So that said, we're in the midst of a really serious curriculum revision, which is on the table and being designed right now, and was voted on by our faculty and improved and

has to go through the school of engineering and get approval. Then there's going to be a little bit of jostling as everything gets worked out. But started next year, you're going to see some changes in the department. Some of these courses are going to get morphed, and they won't be exactly the way they are today.

As a Stanford undergrad, you actually have the option of graduating under any set of requirements that are in play any time you're an undergrad. So you could graduate under this year's requirements, or you could wait. I think for most, the right answer's going to be wait because I think the new arrangement is very much more student-friendly. It has more flexibility in the program.

If you look at the CS program as it is, it has a lot of pick one of two, pick one of three, pick one of two, where you're very constrained on what you can choose. The new program is going to be more track-based where you pick an area of interest. You say, I'm really interested in the graphics. You do a little depth concentration in that, and then just have room for a very wide selection of electives to fill up the program as opposed to a smorgasbord where we forced you to pick through a bunch of different areas.

So for most students, I think it gives you flexibility that actually gives you the options to pick the classes that are most interesting to you and get the material that you want. It's a growing recognition of the fact that CS has really broadened as a field. Some say that our major looks the same way it did ten years ago, a classic definition of here's these seven things that all matter. The things aren't seven anymore. There's no 15 of them, 25 of them.

If you look at the spectrum of things that computer science is now embracing, we can't say, you have to take one of all these things without keeping you here for eight years. So we are allowing that our field's footprint has gotten so large that we need to let you pick and choose a little bit to get the slice that seems most appropriate for where you're headed.

So you'll see more about that. If you are thinking about being a CS major and have not yet declared, I would advise that you add yourself to what's called the considering CS list, and you can get to this from the CS course advisor's page, which, off the top of my head, I think it might be CSadvising, but I'm not positive. I'll put the link on our webpage, when I remember what it is.

It is a mailing list. So we have a mailing list of the declared undergrads, but we also have a list for people who are at least potentially interested in the major and just want to be kept up to date on things that are happening. There's going to be a lot of activity on this list in the upcoming months as we publish the information about the new curriculum. So you'll know what's coming down the pipes so you can make good decisions for preparing for what will be [inaudible], what the transition plan will be as we move into the new courses and how to make sure you land in the right place when all is said and done.

Then I put a little note about other majors that exist. CS is the home base for people who are solving problems using computers. It spans a large variety of people doing a lot of different things. Then there's a couple other majors that we participate in, in the interdisciplinary sense, that UCS is part of a larger context for doing things. So computer systems engineering, which is between us and EE, which is a little bit more about hardware.

The symbolic systems program, which is run by linguistics and includes cooperation from psychology, philosophy, communications, CS looking at artificial intelligence and modeling human thought and understanding intelligence as expressed in that way. Then there's a mathematical computational science program that's homed in the statistic department that is joint between math, computer science and the MS and E. It's more of an applied math degree. So taking mathematical thinking and using computers and statistical things to analyze and think about things. A lot of financial people end up being drawn to that particular one or people who are doing risk analysis or that sort of combination of things.

But where CS has a role to play in all of these because it's a great tool for solving these kind of problems. So different ways to get at different pieces of the major.

I've put a couple notes of things that I think are worth looking into in case they, at some point, are the right thing for you to look at. Section leading, so joining the 106 staff and shepherding the next generation of students through the program is a great way to strengthen your own knowledge. There's nothing like making sure you understand recursion as to try and explain it to someone else who doesn't understand it. I can really do a lot for the personal connection to you and your craft as well as just being involved in a great community. The section leaders are a really wonderful, dedicated group of students who are fun to be around and great people to make friends with. So I highly encourage you to take a look at that.

We interview for that every quarter, so usually about week six, we put out applications and bring people in for interviews. We're always looking for new people to join on a quarterly basis.

There are research opportunities all around the campus. One that the CS department sponsors in particular is their [inaudible] summer institute. So matching professors with undergrads to do work in the summer for slave wages. But great opportunity. You can get started on thinking about how research might be part of your future. One of the best preparations for thinking about grad school.

We do have an honors program. Like most, we have a chance to stake out on an independent project and show your mettle. We have a co-terminal masters program where you can get some of the depth before you leave this place. Not to mention, there's a million fabulous things to do with Stanford, getting involved with arts or nonprofit organizations or sports or the student leadership or whatever.

Maybe the most important memory I have from my undergrad was that the trickiest part of my undergrad was learning what to say no to. Realizing that there are 10,000 fabulous, worthwhile opportunities, and doing all of them only means sacrificing your sanity and the quality of the things you do. So be careful and thoughtful about choosing the things you do and embracing them. Be okay with saying, I won't do this.

Even though it's great to go overseas, and I would've liked that opportunity, it's not going to fit, and I'm okay with that. Don't beat yourself up over it and cause yourself too much grief trying to do too many things.

I won't be here on Friday. Keith is going to come. He's going to teach you about C++ in the real world. I'm going to see you guys Friday at the final. I think that's all I need to say for today.

[End of Audio]

Duration: 51 minutes