ProgrammingAbstractions-Lecture04

Instructor (Julie Zelenski):Hey, good afternoon. Good afternoon. I've gotten a couple emails, which means that there's actually been compiler success, but I've also gotten a couple emails about compiler failures, so just a reminder that one of the tasks that you need to accomplish this week is getting the compiler installed where you're going to be working.

If it's in the clusters, that's actually been done for you, but if you're doing it on your own machine there's just a surprisingly vast array of things that can go wrong along the way that you don't want to be wrestling with at the last minute, so certainly try to get on that soon and then getting in touch with us if you run into some snags so that we can help you get past that and move on to running the code that you need read this week.

We posted a couple announcements on the webpage, one for X-Code and one for Visual Studio, and we will continue to do any updates that we think will be beneficial to the whole class. So do keep an eye on that and look for things that are coming out so that you can avoid things that we already know about and not spend too much time on them.

Sections start this week, but your Section emails should have been emailed to you yesterday, so you should know when and where your Section's meeting, who your Section Leader is. If you have – something's changed, and that makes that completely impossible to work for you we have kind of a limited ability to get you switched into something else, but it's a little bit dependent on where we have space.

So if you need something like that, get back to the sign-up page, there's an add/late option, and that will coordinate with the remaining spaces to get you someplace that works better for you. Today's topic is one of – we're going to talk more about [inaudible] I started that at the beginning of Monday and didn't get very far into it, so I'm gonna go through a little bit more of that today.

And then I'm going to start talking about CS106 class libraries, some of the facilities we're going to be using all quarter long, we're going to introduce today and Friday to get somewhere with those things. Much of this is just covered in the handout. Handout 14, which went out today is this big 20 something page handout, is the kind of reading material for the next two lectures, and then after that we will go back to start reading in the text.

We'll be picking up with Chapter 4 next week after the holiday. Any administration questions on your mind? How many people have actually successfully installed a compiler? Have stuff working – okay, so that's like a third of you, good to know. Remaining two thirds, you want to get on it.

Okay, so we started to talk about this on Monday, and I'm gonna try to finish off the things that I had started to get you thinking about; about how input/output works in C++.

We've seen the simple forms of using stream insertion, the less than less than operator to push things on to cout, the Console Output Stream.

A C-Out is capable of writing all the basic types that are built into C++, ants and doubles and cars and strings, right, by virtue of just sort of putting the string on the left and the thing you want on the right, it will kind of take that thing and push it out onto stream.

You can chain those together with lots and lots of those < < to get a whole bunch of things, and then the endl is the – what's called stream manipulator that produces a new line, starts the next line of text, a line beneath that. The analog to that on the reading side is the stream extraction operator, which is the > >.

And then when applied to an input stream it attempts to sort of take where the cursor position is in the input stream and read the next characters using the expected format given by the type of the thing you're trying to extract. So in this case what I'm saying, CN > > extract an integer here, X being an integer. What it's gonna look for in the input stream is it's going to skip over white space. So by default the stream extraction always skips over any leading white space.

That means tabs, new lines, and ordinary space characters. So scans up to that, gets to the first non-space character and then starts assuming that what should be there is a number, and so number being, a sequence of digit characters. And in this case, because it's integer, it shouldn't have a dot or any of the exponentiations sort of things that a real number would.

If it runs into something that's not integer, it runs into a character, it runs into a punctuation, it runs into a 39.5, what happens is that the screen goes into a fail state where it says, I – you told me to expect an integer. What I read next wasn't an integer. I don't know how to make heads or tails of this. So it basically just throws up its hand.

And so it – at that point the stream is – it requires you to kind of intervene, check the fail state, see that something's wrong, clear that fail state, decide what to do about it, kind of restart, and kind of pick up where you left off. It makes for kind of messy handling to have all that code kind of in your face when you're trying to do that reading, and that's actually why we've provided the things like get integer, get line and get wheel, and the simple I/O library to just manage that for you.

Basically what they're doing is in a loop they're trying to read that integer off the console. And if it fails, write resetting the stream, going back around asking the user to type in – give it another try, until they get something that's well formed. So typically we're just going to use these, because they just provide conveniences.

You could certainly use this, but it would just require more effort on your part to kind of manage the error conditions and retry and whatnot. So that's why it's there. The C++ file I/O; so the console is actually just a particular instance of the stream. Cout and cin are the string that's attached to the users interface console there.

That the same sort of mechanism is used to read files on disks, so text files on disks that have contents you like to pull into a database, or you want to write some information out to a file, you use the file stream for that. There is a header called fstream, standard C++ header in this case, so enclosed in < >, that declares the isstream and the osstream.

The input file stream for reading, the output file stream for writing. Declaring these variables; this [inaudible] just sets up a default stream that is not connected to anything on disc. Before you do anything with it you really do need to attach it to some named location, some file by name on your disk to have the right thing happen, to read from some contents, or to write the contents somewhere.

The operation that does that is open, so the isstream and the osstream are objects, so dot notation is used to send messages to it. In this case, telling the input stream to open the file whose name is "names.txt." The behavior for open is to assume that you meant the file in the current directory if you don't otherwise give a more fully specified path.

So this is almost always the way we're going to do this, we're just going to open a file by name. It's going to look for it in the project directory, where your code is, where you project is, so kind of right there locally. Now this will look for a file whose name is exactly names.txt, and then from that point the file positions, the kind of cursor we call it, is positioned at the beginning of the input stream.

The first character read will be the first character of names.txt, and as you move forward it will read its way all the way to the end. Similarly, doing an outopen, it opens a file and kind of positions the writing at the very beginning that will – the first character written will be the first character then when you finish.

And that file, they'll be written in sequence. So this is one of those places, actually, probably the only one that this direction is going to be relevant for. I talked a little bit last time about C-strings and C++ strings, and you might have been a little bit worried about why I'm telling you you need to know that both exist.

And so last time I talked a little about one way in which C-strings don't do what you think, in that one case of concatenation, and how you can do a – force a conversion from the old to the new. Now, I also mentioned that there was a conversion that went in the opposite direction. You had a new string, and you wanted the old one.

And one of the first questions you might ask is well why would I ever want to do that? Why would I ever want to go backwards? Why do I want to move back to the older yucky thing? This is the case that comes up; the open operation on isstream and osstream expects its argument to be specified as an old style string. This is actually just an artifact; it has to do with it – the group that was working on designing the string package. The group that was designing the string package were not in sync, and they were not working together.

The string package was finalized before the string package was ready and so it depended on what was available at the time and that was only the old style string. So as a result, it wants an old style string, and that's what it takes, and you can't give it a C++ string. So in double quotes – so this is the case where the double quotes are actually old style strings, in almost all situations gets converted on your behalf automatically. In this case it's not being converted and it's exactly what's wanted.

So if you have a name that's a string constant or a literal, you can just pass it in double quotes to open. If you have a C++ variable, so you've asked the user for what file to open, and you've used getline to read it into a string, if you try to pass that C++ string variable to open, it will not match what it's expecting.

I do need to do that conversion asking it to go .c_str to convert itself into the old style format. So that was sort of where I was getting to when I kind of positioned you to realize this was gonna someday come up. This is the one piece of the interface that will interact with this quarter that requires that old string, where you'll have to make that effort to convert it backwards.

Both of these operations can fail. When you open a file and [inaudible] – question here?

Student: So how hard [inaudible]?

Instructor (Julie Zelenski):You know it's obviously extremely easy to do it; the issue has to do with compatibility. They announced it this way, people wrote code that expected it this way and then you change it out from under them and all this code breaks that used to work. And so as a result of this [inaudible] compatibility an issue of once we kind of published it and we told people this was how it works, we can't really take it away from them.

And so part of that's – sort of part of what we're doing within C++2, which is things that used to work in C still need to work in C, and so as a result there's a certain amount of history that we're all carrying forward with us in a very annoying way. I totally agree that it seems like we could just fix it, but we would break a lot of code in the process and anger a lot of existing programmers.

So both of these open calls could fail; you might be able to – try to open a file and it doesn't exist, you don't have the permissions for it, you spelled the name wrong. Similarly trying to open it for writing, it's like you might not have write permission in the directory. And in either situation you need to know, well did it open or did it not?

There's not a return value from open that tells you that. What there is is a member function called .fail, that you can ask the stream at any point, are you in a fail state. So for operations that actually kinda have a chance of succeeding or failing in the string, you'll tend to actually almost write the code as a try it then check in .sale. So try to read this thing, check in .sale. Try to open this file check in .sale as your way of following up on did it work and making sure that you have good contents before you keep going.

If the in .open has failed, then every subsequent read on it will fail. Once the string is in a fail state, nothing works. You can't read or write or do anything with it until you fix the error, and that's the in .clear command that kind of resets the state back into a known good state, and then you have a chance to retry.

So for example, if you were trying to open a file that the user gave you a name for, they might type the name wrong. So you could try in .openit, check in .dot fail. If it failed, say no, no, I couldn't open that file, why don't you try again, get a new name, and then you'd clear the state, come back around and try another in .open to – until you get one that succeeds.

Once you have one of those guys open for reading or writing, there are three main ways that you can do your input/output. We have seen this form a little bit, this one with the insertion/extraction, these other two are more likely to be useful in the file reading state as opposed to interacting with the user state, and they have to deal with just breaking down the input more fine graindly.

Let's say this first one is reading and writing single characters. It might be that all I want to do is just go through the file and read it character by character. Maybe what I'm trying to write is something that will just count the characters and produce a frequency count across the file, tell me how many A's and B's and C's are in it, or just tell me how many characters are in the file at all.

In .get is the number function that you send to an input file stream that will retrieve the next character. If [inaudible] the next character from the stream it returns EOF when there are no more characters. EOF is the end of file marker, it's actually capital EOF, it's the constant that's defined with the class. And so you could read till EOF as a way of just getting them character by character.

Similarly there is a put on the other side, which is when you're writing, do you just want to write a single character. You could also do this with out << ch, which writes the character. This actually just does a put of the character, just kind of a matching function in the analog to get input that do single character io.

Sometimes what you're trying to do is process it line by line. Each line is the name of somebody and you're kind of putting those names into a database. You don't want to just assemble the characters by characters, and you don't know how many tokens there might be, that the white space might be that there's Julie Diane Zelenski, sometimes there might be Julie Zelenski, you don't know how many name pieces might appear to be there.

You can use getline to read an entire line in one chuck. So it'll read everything up to the first new line character it finds. It actually discards the new line and advances past it. So what you will get is – the sequence of characters that you will have read will be everything up to and not including the new line. The new line will be consumed though so that reading will pick up on the next line and go forward.

Getline is a free function. It is not a member function on the stream. It takes a stream as its first argument. It takes a string by reference as its second argument, and it fills in the line with the text of the characters from here to the next line read in the file.

If it fails the way you will find out is by checking the fail states. You can do a getline inline and then in .fail after it to see, well did it write something in the line that was valid? If it failed, then the contents of line are unchanged, so they'll be whatever nonsense they were. So it's a way of just pulling it line by line.

This name has the same words in it as rgetlineGL in the sympio, which shows that it's kind of a reasonable name for the kind of thing that reads line by line, but there is a different arrangement to how it's – what it's used for and how it's it used. So rgetline takes no arguments and returns a line read for the console. The lower case getline takes the file stream to read from and the string to write it into and does not have a return value.

You check in .fail if you want to know how it went. So write the entire line out there, [inaudible] a put line equivalence, so in fact you could just use the out stream insertion here, stick that line back out with an nline to kind of reproduce the same line your just read.

And then these we've talked a little about, this idea of formatted read and write, where it's expecting things by format. It's expecting to see a character, it's expecting to see an integer, and it's expecting to see a string. It uses white space as the default delimiter between those things.

So it's kind of scanning over white space and discarding it and then trying to pull the next thing out. These are definitely much trickier to use because if the format that you're expecting doesn't show up, it causes the stream to get new fail state, and you have to kind of fix it and recreate it.

So often even when you expect that things are going to be, let's say, a sequence of numbers or a name fall by number, you might instead choose to pull it as a string and then use operations on the string itself to kinda divide it up rather than depending on stream io because stream io is just a little bit harder to get that same effect.

And then in all these cases write in .fail. There is also – you could check out.fail. It's just much less common that the writing will fail, so you don't see it as much, but it is true for example, if you had wanted a disk space and you were writing, a write operation could fail because it had wanted a space or some media error had happened on the disk, so both of those have reasons to check fail.

So let me do just a little bit of live coding to show you that I – it works the way I'm telling you. Yeah?

Student: So the fail function, is it going to always be the stream that's failing and not the function that's failing?

Instructor (Julie Zelenski):Yes, pretty much. There are a couple rare cases where the function actually also tells you a little bit about it, but a general fail just covers the whole general case of anything I have just got on the stream fail so any of the operations that could potentially run into some error condition will set the fail in such a way that your next call to in .fail will tell you about it.

And so that's the – the general model will be; make the call, check the fail, if you know that there was a chance that something could have gone wrong and then you want to clean up after it and do something [inaudible].

So I'm gonna show you that I'm gonna get the name of the file from the user here, I'm going to use in .open of that, and I'm going to show you the error that you're gonna get when you forget to convert it, while I'm at it. And then I'll have like an in .fail error wouldn't – file didn't open. First I just want to show you this little simple stuff; I've got my ifstream declared, my attempt to open it and then my check for seeing that it failed.

I'm gonna anticipate the fact that the compiler's gonna be complaining about the fact that it hasn't heard about fstream, so I'm gonna tell it about fstream. And I'm gonna let this go ahead in compiling, although I know it has an error in it, because I want to show you sort of the things that are happening.

So the first thing it's complaining about actually is this one, which is the fact that getline is not declared in the scope, which meant I forgot one more of my headers that I wanted. Let me move this up a little bit because it's sitting down a little far. And then the second thing it's complaining about is right here.

This is pretty hard to see, but I'll read it to you so you can tell what it says; it says error, there's no matching function call and then it has sort of some gobbly gook that's a little bit scary, but includes the name ifstream. It's actually – the full name for ifstream is a lot bigger than you think, but it's saying that there's – the ifstream is open, and it says that it does not have a match to that, that there is no open call on the ifstream class, so no member function of the ifstream class whose name is open, whose argument is a string.

And so that cryptic little bit of information is gonna be your reminder to jog your memory about the fact that open doesn't deal in the new string world, it wants the old string world. It will not take a new string, and I will convert it to my old string, and then be able to get this thing compiling.

And so when it runs if I enter a file name of I say [inaudible], it'll say error file didn't open, some file that I don't have access for. It happens that I have one sitting here, I think, whose name is handout.txt. I took the text of some handout and then I just left it there. So let me doing something with that file. Let's just do something simple where we just count the number of lines in it. Let's say – actually I'll make a little function that – just to talk a little bit about one of the things that's a little quirky about ifstreams is that when you pass an ifstream you will typically want to do so by reference.

Not only is this kind of a good idea, because the ifstream is kind of changing in the process of being read. It's updating its internal state and you want to be sure that we're not missing this update that's going on. It's also the case that most libraries require you to pass it by reference. That it doesn't have a model for how to take a copy of a stream and make another copy that's distinct. That it really is always referring to the same file, so in fact in most libraries you have to pass it by reference.

So I'll go ahead and pass it by reference. I'm gonna go in here and I'm just gonna do a line-by-line read and count as I go. I'm gonna write this as a wild [inaudible], and I'm gonna say read the next line from the file into the variable, and then if in .fail – so if it was unable to read another line, the – my assumption here is gonna be that we're done, so it will fail as eof . It's the most common reason it could fail. It could also fail if there was some sort of more catastrophic error, you're leading a file from a network and the network's gone down or something like that.

In our case its right, the in .fail is going to tell us yeah, there's nothing more to read from this file, which means we've gotten to the end. We've advanced the count. Whenever we get a good line we go back around, so we're using kind of the wild true in this case because we have a little bit of work to do before we're ready to decide whether to keep going, in this case, reading that line.

And then I return the count at the end, and then I can then down here print it nom lines = mi call to count lines of n and l. Okay. Let me move that up a little bit. Last time I posted the code that I wrote in the editor here, and I'll be happy to do that again today, so you shouldn't need to worry about copying it down, I will post it later if you want to have a copy of it for your records, but just showing, okay, yeah, we're just a line by line read, counting, and then a little bit more of the how do you open something, how do you check for failure.

And when I put this together, what does it complain about? Well I think it complains about the fact that I told it my function returned void, but then I made it return it. And that should be okay now. And so if I read the handout.txt file, the number of lines in it happens to be 28. It's just some text I'd cut out of the handout, so there are 28 new line characters is basically what it's telling me there.

So I can just do more things, like I could use – change this loop and instead use like get to do a single character count. I could say how many characters were in there. If I used the tokenization and I said, well just tell how many strings I find using string extraction, it would kind of count the number of non-space things that it found and things like that.

Typically the IO is one of those errors I said where there's like a vast array of nuances to all the different things you can do with it, but the simple things actually are usually fairly easy, and those are the only ones that really going to matter to us as being able to do a little bit of simple reading and file reading/writing to get information into our programs.

How do you feel about that? Question?

Student: Sorry, why do have getline an empty string?

Instructor (Julie Zelenski):So getline, the one that was down here? This one?

Student: No, the one that –

Instructor (Julie Zelenski):Oh, the one that's up here. So yeah, let's talk about that. The getline that's here is – the second argument to getline is being passed by reference, and so it's filling in that line with the information it read from the file. So I just declared the variable so I had a place to store it and I said, okay, read the next line from the file, store the thing you read in the line.

It turns out I don't actually care about that information, but there's no way to tell getline to just throw it away anyway.

Student: Oh.

Instructor (Julie Zelenski):So I'm using it to just kinda move through line-by-line, but it happens to be that getline requires me to store the answer somewhere, and I'm storing it. Instead of returning it, it happens to use the design where it fills it in by reference. There's actually – it turns out to be a little bit more efficient to do a pass by reference and fill something in, then to return it. And the C++ libraries in general prefer that style of getting information back out of a function as opposed to the function return, which you think of as being a little more natural design.

There's a slight inefficiency to that relative to the pass by reference and the libraries tend to be very hyper-conscious of that efficiency, so they tend to prefer this slightly more awkward style. Question?

Student: Why in the main [inaudible] does the error open [inaudible] file didn't open with [inaudible] like print error: file didn't open?

Instructor (Julie Zelenski):You know it's just the way that error works. Error wants to make sure that you don't mistake what it does, and so it actually prefixes whatever you ask it to write with this big ERROR in uppercase letters, and so the purpose of error is twofold; is to report what happened and to halt processing.

And so when it reports that it actually prefixes it with this big red E-R-R-O-R just to say don't miss this, and then it halts processing there. And it's just – the error [inaudible] libraries function, which is your way of handling any kind of catastrophic I can't recover from this. And it's certainly something we don't want anybody to overlook, and so we try to make it really jump out at you when it tells you that.

Student: So this is in symbio?

Instructor (Julie Zelenski):It is in genlib actually.

Student: Oh.

Instructor (Julie Zelenski):So error's actually declared out of genlib.

Student: And can we use it – so it's global basically?

Instructor (Julie Zelenski):It is global. It's a telefree function, and you will definitely have occasion to use it. Right, it's just – it's your way of saying something happened that there's just no recovery from and continuing on would not make sense. Here's a – stop and help and alert the user something's really wrong, so you don't want to keep going after this because there's no way to kind of patch things back together.

In this case probably a more likely thing we'd do, is I should say give me another name, let's go back around and try again, would be a sort of better way to handle that. I can even show you how I would do that. I could say, well while true, enter the name, and maybe I could change this to be well if it didn't fail then go ahead and break out of the loop. Otherwise, just report that the file didn't open, and say try again.

And then the last thing I will need to do is clear that state. So now it's prompting, trying to open it. If it didn't fail it will break and then it will move forward to counting the lines. If it did fail it'll continue on through here reporting this message, and then that clear, very important, because that clear kind of gets us back in the state where we can try again.

If we don't clear the error and we try to do another in .open, once the string is in a fail state it stays in a fail state until you clear it, and no subsequent operation will work whatsoever. It's just ignoring everything you ask it to do until you have acknowledged you have done something about the problem, which in this case was as simple as clearing and asking to open again.

So if I do it this way I enter some name it'll say that didn't open, try again. And then if I say handout.txt, it'll open it and go ahead and read. All right, any questions about iostreams? We're gonna move away from this [inaudible], if there's anything about it you'd like to know I'd be happy to answer it.

So let me get us back to our slides, and I'll kind of move on to the more object-oriented features of the things we're going to be depending on and using this quarter. So the libraries that we have been looking at, many of them are just provided as what we call free functions. Global functions that aren't assigned to a particular object, they are part of a class, so asking for random integer, reading a line, competing the square root, gobs of things are there that just kind of have functionality that you can use anywhere and everywhere procedurally.

We've just started to see some things that are provided in terms of classes, the string of the class, that means that you have string objects that you're messaging and having them manipulate themselves. The stream object also is class, ifstream, ofstream, those are all

classes that you send messages like open to and fail to, to ask about that streams state or reset its state.

This idea of a class is one that's hopefully not new to you. Most of you are coming from Java have – this is pretty much the only mechanism for writing code for Java is in the context of a class. Those of you who haven't seen that as much, we're going to definitely be practicing on this in our – some simple things you need to know to kind of just get up to the vocabulary wise is class is just a way of taking a set of fields or data and attaching operations to it to where it kind of creates a kind of an entity that has both its state and its functionality kind of packaged together.

So in the class interface you'll say here is a time object, and a time object has an hour and a minute and you can do things like tell me if this time's before that time or what the duration starting at this time and this end time would – there would be all these behaviors that are like [inaudible] to do. Can you print a time, sure. Can I read a time for a file, sure.

As long as the interface for the time class provides those things, its kinda this fully flip – fleshed out new data type that then you use time objects of whenever you need to work with time. The idea is that the client use the object, which is the first role we're gonna be in for a couple weeks here, is you learn what the abstraction is. What does the class provide? It provides the notion of a sequence of characters, that's what stream does.

And so that sequence has all these operations; like well tell me what characters are at this position, or find this sub-string, or insert these characters, remove those characters. And internally it's obviously doing some machinations to keep track of what you asked it to do and how to update its internal state.

But what's neat is that from the outside as a client you just think well there's a sequence of characters there and I can ask that sequence of characters to do these operations, and it does what I ask, and that I don't need to know how it's implemented internally. What mechanisms it uses and how it responds to those things to update it state is very much kind of behind the abstraction or inside that black box, sometime we'll call it to kind of suggest to ourselves that we can't see inside of it, we don't know how it works. It's like the microwave, you go up and you punch on the microwave and you say cook for a minute. Like what does the microwave do? I don't know, I have no idea, but things get hot, that's what I know. So the nice thing about [inaudible] is you can say, yeah, if you push this button things get hot and that's what I need to know. [Inaudible] has become widely industry standard in sort of all existing languages that are out there. It seems like there's been somebody who's gone to the trouble of trying to extend it to add these object [inaudible] features and languages like Java that are fully object oriented, are very much all the rage now. And I thought it was interesting to take just a minute to talk about well why is it so successful? Why is object oriented like the next big thing in programming? And there are some really good valid reasons for why it is a very sensible approach to writing programs that is worth thinking a little bit about. Probably the largest sort of motivation for the industry has to do with this idea of taming complexity that certainly

one of the weaknesses of ourself as a discipline is that the complexity kinda can quickly spiral out of control. The programs that – as they get larger and larger, their interactions get harder and harder to model and we have more and more issues where we have bugs and security flaws and viruses and whatnot that exploit holes in these things. That we need a way as engineers to kind of tighten down our discipline and really produce things that actually don't have those kind of holes in them. And that object oriented probably means one of the ways to try to manage the complexities of systems. That instead of having lots and lots of code that [inaudible] things, if you can break it down into these objects, and each class that represents that object can be designed and tested and worked on independently, there's some hope that you can have a team of programmers working together, each managing their own classes and have them be able to not interfere with each other too much to kind of accomplish – get the whole end result done by having people collaborate, but without them kind of stepping on top of each other. It has a – the advantage of modeling the real world, that we tend to talk to talk about classes that kind of have names that speak to us, what's a ballot, what's a class list, what's a database, what is a time, a string, that – a fraction? These things kind of – we have ideas about what those things are in the real world, and having the class model that abstraction makes it easier to understand what the code is doing and what that objects role is in solving the problem.

It also has the advantage of [inaudible] use. That once you build a class and it's operations, the idea is that it can be pulled out of the – neatly out of the one program and used in another if the design has been done, and can be changed extended fairly easily in the future if the design was good to begin with.

So let me tell you what kind of things we're going to be doing in our class library that will help you to kind of just become a big fan of having a bunch of pre-written classes around. We have, I think, seven classes – I think there's eight actually in our class library that just look at certain problems that either C++ provides in a way that's not as convenient for us, or is kind of missing, or that can be improved on where we've tackled those things and given you seven classes that you just get to use from the get go that solve problems that are likely to come up for you.

One of them is the scanner, which I kind of separated by itself because it's a little bit of an unusual class, and then there's a bunch of container classes on that next line, the vector grid, staque, math and set that are used for storing data, different kinds of collections, and they differ in kind of what their usage pattern is and what they're storing, how they're storing it for you.

But that most programs need to do stuff like this, need to store some kind of collection of date, why not have some good tools to do it. These tools kinda let you live higher on the food chain. They're very efficient, they're debugged, they're commented, the abstraction's been thought about and kind of worked out and so they provide kinda this very useful piece of function [inaudible] kinda written to you ready to go.

And then I – a little note here is that we study these – we are going to study these abstractions twice. We're gonna look at these seven classes today and Friday as a client, and then start using them all through the quarter. In about a week or so after the mid-term we're gonna come back to them and say, well how are they implemented?

That after having used them and appreciated what they provided to you, it will be interesting, I think, to open up the hood and look down in there and see how they work. I think this is – there is an interesting pedagogical debate going on about this, about whether it's better to first know how to implement these things and then get to use them, or to use them and then later know how to implement them.

And I liken it to a little bit if you think about some things we do very clearly one way or the other in our curriculum, and it's interesting to think about why. That when you learn, for example, arithmetic as a primary schooler, they don't give you a calculator and say, here, go do some division and multiplication, and then later try to teach you long division. You'll never do it.

You'll be like, why would I ever do this, this little box does it for me, the black box. So in fact they drill you on your multiplication tables and your long division long before they let you touch a calculator, which I think is one way of doing it.

And, so – and for example, it's like we could do that with you, make you do it the kind of painful way and then later say, okay, well here's these way you can avoid being bogged down by that tedium. On the other had, think about the way we teach you to drive.

We do not say, here's a wheel and then they say, let me tell you a little bit about the combustion engine, you know, we give you some spark plugs and try to get you to build your car from the ground up. It's like you learn to drive and then if you are more interested in that you might learn what's under the hood, how to take care of your car, and eventually how to do more serious repairs or design of your own care.

Where I think of that as being a client first model, like you learn how to use the car and drive and get places and then if it intrigues you, you can dig further to learn more about how the car works. So that's definitely – our model is more of the drive one than the arithmetic one that it's really nice to be able to drive places first.

Like if I – we spent all quarter learning how to build a combustion engine and you didn't get to go anywhere, I'd feel like you wouldn't have tasted what – where you're trying to get, and why that's so fabulous.

So we will see them first as a client, and you'll get to do really neat things. You'll discover this thing called the map where you can put thousands, millions of entries in and have instantaneous look-up access on that. That you can put these things in a stack or a queue and then have them maintained for you and popped back out and all the storage of that being managed and the safety of that being managed without you having to kinda take any active role in that.

That they provide functionality to you, that you just get to – leverage from the get go, and hopefully it will cause you to be curious though, like how does it work, why does it work so well, and what kind of things must happen behind the scenes and under the hood so that when we get to that you're actually kind of inspired to know how it did it, what it did.

So I'm gonna tell you about the scanner and maybe even tell you a little bit about the vector today, and then we'll do the remaining ones on Friday, perhaps even carrying over a little bit into the weeks to get ourselves used to what we've got.

The scanner I kind of separated because the scanner's more of a task based object then it is a collection or a container for storing things. The scanner's job is to break apart input into tokens. To take a string in this case that either you read from the file or you got from the user, or you constructed some way, and just tokenize it.

It's called tokenizer parsec. That this is something a little bit like – strained extraction kind of does this, but strained extraction, as I said, isn't very flexible, that it doesn't make it easy for you to kind of – you have to sort of fully anticipate what's coming up on the string. There's not anyway you can sort of take a look at it and then to decide what to do with it and decide how to change your parstring strategy.

And scanner has a kind of flexibility that lets it be a little bit more configurable about what you expect coming up and how it works. So the idea is that basically it just takes your input, you know, this line contains ten tokens, and as you go into a loop saying, give me the next token, it will sub-string out and return to you this four character string followed by this single character space and then this four character line and space, and so the default behavior is to extract all the tokens to come up, to use white-space and punctuation as delimiters.

So it will kind of aggregate letters and numbers together and then individual spaces and new lines and tabs will come out as single character tokens. The parenthesis and dots and number signs would all come out as single character tokens, and it just kind of divides it up for you.

Okay. It has fancy options though that let you do things like discard those face tokens because you don't care about them. To do things like read the fancy number formats. So it can read integer formats and real formats, it can do the real format with exponentiation in it with leading minus', things like that, that you can control with these setters and getters, like what it is you wanted to do about those things.

You can it things like when I see an opening quote, I want you to gather everything to the closing quote, and so it does kind of gather phrases out of sequence if that's what you want. And so you have control over when and where it decides to do those things that lets you kind of handle a variety of kind of parsing and dividing tasks by using the scanner to get that job done.

So I listed some things you might need, if you're reading txt files, you're parsing expressions, you were processing some kind of commands, that this scanner is a very handy way to just divide that [inaudible] up.

You could certainly do this kind of stuff manually, for example, like using the find on the string and finding those faces and dividing it up, but that the idea is just doing that in a more convenient way for you than you having to handle that process manually.

This is what its interface looks like. So this is a C++ class definition. It looks very similar to a Java class definition, but there's a little bit of variation in some of the ways the syntax comes through in the class. The class being here is scanner, the public colon introduces a sequence of where everything from here until the next access modifier is public. So I don't actually have public repeated again and again on all the individual entries here.

It tells us that the scanner has a constructor that takes no arguments; it just initializes a new empty scanner. I'm gonna skip the destructor for a second; I'll come back to it. There is a set input member function that you give it the string that you want scanned and then there's these two operations that tend to be used in a look where you keep asking are there more tokens and if so, give me the next token, so it just kind of pulls them out one by one.

I picked just one of the space – of the particular advanced options to show you the format for them. There's actually about six more that deal with some other more obscure things. This one is how is it you'd like it to deal with spaces, when you see face tokens, should they be returned as ordinary tokens or should you just discard them entirely and not even bother with them?

The default is what's called preserve spaces, so it really does return them, so if you ask and there's only spaces left in the file, it will say there are more tokens and as you call the next token we'll return those spaces as individual tokens.

If you instead have set the space option of ignore spaces, then it will just skip over all of those, and if all that was left in the file was white space when you ask for more tokens, it will say no. And when you ask for a token and there's some spaces leading up to something it will just skip right over those and return the next non-space token.

There's a variety of these other ones that exist that handle the floating point and the double quote and other kind of fancy behaviors. There's one little detail I'll show you that's a C++ ism that isn't – doesn't really have a Java analog, which is the constructor which is used as the initialization function for a class has a corresponding destructor.

Every class has the option of doing this. That is the – kind of when the object is being created, the constructor is being called. When the object is being de-allocated or destroyed, going out of scope, the destructor is called. And the pairing allows sort of the

constructor to do any kind of set up that needs to be done and the destructor to do any kind of tear down that needs to be done.

In most cases there's not that much that needs to be there, but it is part of the mechanism that allows all classes to have an option kind of at birth and death to do what it needs to do. For example, my file stream object, when you – when it goes away, closes it file automatically.

So it's a place where the destructor gets used to do cleanup as that object is no longer valid. So a little bit of scanner code showing kind of the most common access pattern, is you declare the scanner. So at this point the scanner is empty, it has no contents to scan.

Before I start pulling stuff out of it, I'm typically gonna call a set input on it, passing some string. In this case the string I'm passing is the one that was entered by the user, using getline. And then the ubiquitous loop that says well while the scanner has more tokens, get the next token.

And in this case I'm not even actually paying attention to what those tokens are, I'm just counting them. So this one is kind of a very simple access that just says just call the next token as many times as you can until there are no more tokens to pull out. Way in the back?

Student:[Inaudible] I mean, like in the beginning when it says scanner, scanner, do we write scanner scanner = new scanner () or [inaudible]?

Instructor (Julie Zelenski):Yes. Not exactly. So that's a very good example of like where Java and C++ are gonna conspire to trip you up just a little bit, that in Java objects were always printed using the syntax of new. You say new this thing, and in fact that actually does an allocation out in what's called the heap of that object and then from there you use it.

In C++ you actually don't have to put things in the heap, and in fact we will rarely put things in the heap, and that's what new is for. So we're gonna use the stack to allocate them. So when I say scanner scanner, that really declares a scanner object right there and in this case there are no [inaudible] my constructor, so I don't have anything in parenths.

If there were some arguments I would put parenths and put the information there, but the constructor is being called even with out this new. New actually is more about where the memory comes from. The constructor is called regardless of where the memory came from.

And so this is the mechanism of C++ to get yourself an object tends to be, say the class name, say the name of the variable. If you have arguments for the constructor, they will go in parenths after the variable's name.

So if scanner had something, I would be putting it right here, open parenth, yada, yada. So that's a little C++/Java difference. Oh, that's good. Question over here?

Student: When do we have to use the destructor?

Instructor (Julie Zelenski):So typically you will not ever make a call that explicitly calls the destructor. It happens for you automatically. So you're – [inaudible] you're gonna see it in the interface as part of the completeness of the class it, here's how I set up, here's how I tear down. When we start implementing classes we'll have a reason to think more seriously about what goes in the destructor. But now you will never explicitly call it.

Just know that it automatically gets called for you. The constructor kinda gets automatically called; the destructor gets automatically called, so just know that they're there. One of the things that's – I just want to encourage you not to get too bogged down in is that there's a lot of syntax to C++.

I'm trying to give you the important parts that are going to matter early on, and we'll see more and more as we go through. Don't let it get you too overwhelmed, the feeling of it's almost but not quite like Java and it's going to make me crazy. Realize that there's just a little bit of differences that you kinda got to absorb, and once you get your head around them actually you will find yourself very able to express yourself without getting too tripped up by it.

But it's just at the beginning I'm sure it feels like you've got this big list of here's a thousand things that are a little bit different that – and it will not be long before it will feel like your native language, so hang in there with us.

So I wanted to show you the vector before we get done today and then we'll have a lot more chance to talk about this on Friday. That the other six classes that come in [inaudible] class library are all container classes.

So containers are these things like they're buckets or shells or bags. They hold things for you. You stick things into the container and then later you can retrieve them. This turns out to be the most common need in all programs. If you look at all the things programs do, [inaudible] manipulating information, where are they putting that information, where are they storing it?

One of the sorts of obvious needs is something that is just kind of a linear collection. I need to put together the 100 student that are in this class in a list, well what do I do – what do I use to do that? There is a build in kind of raw array, or primitive array in C++. I'm not even gonna show it to you right now.

The truth is it's functional, it does kinda what it sets out to do, but it's very weak. It has constraints on how big it is and how it's access to it is. For example, you can make an

array that has 10 members and then you can axe the 12th member or the 1,500th member without any good error reporting from either the compiler or the runtime system.

That it's designed for kind of to be a professional's tool and it's very efficient, but it's not very safe. It doesn't have any convenience attached to it whatsoever. If you have a – you create a ten number array and later you decide you need to put 12 things into it, then your only recourse is to go create a new 12 number array and copy over those ten things and get rid of your old array and make a totally new one, that you can't take the one you have and just grow it in the standard language.

So we'll come back to see it because it turns out there's some reasons we're gonna need to know how it works. But for now if you say if I needed to make a list what I want to use is the vector. So we have a vector class in our class library that just solves this problem of you need to collect up this sequence of things, a bunch of scores on a test, a bunch of students who are in a class, a bunch of name that are being invited to a party.

And what it does for you is the things that array does but with safety and convenience built into it. So it does bounds checking. If you created a vector and you put ten things into it, then you can ask for the zero through 9th entries, but you cannot ask for the 22nd entry, it will raise an error and it will use that error function, you will get a big red error message, you will not bludgeon on unknowingly.

You can add things and insert them and then remove them. So I can go into the array and say I'd like to put something in slot zero, it will shuffle everything over and make that space. If I say delete the element that's at zero it will move everything down. So it just does all this kind of handling of keeping the integrity of the list and its ordering maintained on your behalf.

It also does all the management of how much storage space is needed. So if I put ten things into the vector and I put the 11th or the 12th or the – add 100 more, it knows how to make the space necessary for it. Behind the scenes it's figuring out where I can get that space and how to take care of it.

It always knows what count it has and what's going on there, but its doing this on our behalf in a way that that rawray just does not, that becomes very tedious and error prone if it's our responsibility to deal with it. So what the vector is kind of running, it's an instruction. And this is a key word for us in things that we're going to be talking about this quarter is that what you really wanted was a list.

I want a list of students and I want to be able to put it in sorted order or find this person or print them. The fact that where the memory came from and how it's keeping track of is really a tedious detail that I'd rather not have to deal with.

And that's exactly what the vector's gonna do for you, is make it so you store things and the storage is somebody else's problem. You use a list, you get an abstraction. How that – there's one little quirk, and this is not so startling to those of you who have worked on a

recent version of Java, is in order to make the vector generally useful, it cannot store just one type of thing.

That you can't make a vector that stores [inaudible] and service everyone's needs, that it has to be able to hold vectors of doubles or vectors of strings or vectors of student structures equally well. And so the way the vector class is actually supplied is using a feature in the C++ language called templates where the vector describes what it's storing using a placeholder.

It says, well this is a vector of something and when you put these things in they all have to be the same type of thing and when you get one out you'll get the thing you put in, but I will not commit to, and the interface saying it's always an integer, it's always a double. It's left open and then the client has to describe what they want when they're ready to use it.

So this is like the Java generics. When you're using an array list you said, well what kind of things am I sticking in my array list, and then that way the compiler can keep track of it for you and help you to use it correctly.

The interpart of this kinda looks as we've seen before. It's a class vector, it has a constructor and destructor and it has some operations that return things like the number of elements that you can find out whether it has zero elements, you can get the element at index, you can set the element at index, you can add, insert and remove things within there.

The one thing that's a little bit unusual about it is that every time it's talking about the type of something that's going into the vector or something that's coming out of the vector, it uses this elem type which traces its origin back to this template header up there, that is the clue to you that the vector doesn't commit to I'm storing ants, I'm storing doubles, I'm storing strings, it stores some generic elem type thing, which went the client is ready to create a vector, they will have to make that commitment and say this vector is gonna hold doubles, this vector is gonna hold ants, and from that point forward that vector knows that the getat on a vector of ants returns something of n type.

And then add on a vector of nts expects a perimeter of n type, which is distinct from a vector of strings or a vector of doubles. So I'll show you a little code and we'll have to just really talk about this more deeply on Friday. A little bit of this in text for how I make a vector of [inaudible] how I make a vector of strings, and then some of the things that you could try to mix up that the template will actually not let you get away with, mixing those types.

So we'll see this on Friday, so don't worry, there will be time to look at it and meanwhile good luck getting your compiler running.

[End of Audio]

Duration: 52 minutes