ProgrammingAbstractions-Lecture03

Instructor (Julie Zelenski):It's good to see you guys. Just a note about the handouts that are going out today so you aren't confused about our ability to count in sequence – six, eight, nine and ten are in the lobby. Five and seven we're going to bring at the end of the lecture, and I'm trying to avoid killing too many trees by what we do with five and seven. Five and seven are the handouts that tell you about the installing of the compilers and the debuggers of the two compilers, and there's two versions of them. There's a Mac version and a Windows version for the X code versus Visual Studio.

We printed an estimated amount of how many people we thought were going to be using the PCs and how many using the Mac, and so when we bring those in at the end and you're picking them up on the way out, be sure to get the one that you need and not take both. Take the ones for the product you're going to use. If you happen to believe you're really going to use both, it's fine to take both, but I figure that's pretty rare. If you know what platform you're on, take the one for yours and not both.

Those are the ones that tell you about how to get your compiler set up, which seems to be something a couple people this weekend were ready to get a jump on. Assignment 1 is also one of the handouts that's going out today. For assignment one, just to kind of come up to speed is that the first assignment here is not some big, complicated program. It's actually a set of small programs. Each is less than a page of code. They're designed to exercise some skills in isolation – learning how loops and variables work, learning how our graphics library works, and learning how strings work in C++ and files.

The more complicated part is going to come from learning how to express yourself in C++. For those of you who are a little unsure about whether your background is good and whether you're in the right place, this is actually a good testing ground. You should look at those problems and say oh, if I were solving this in my native language, I would be totally able to write this out without even stopping to think much. In C++, I'm definitely going to have some new things about learning how to convert what I know to how it works in C++.

If you do find these challenging to write, that's probably a sign that you're a little bit ahead of you. If you find them completely trivial and way too easy, you may want to think about 106X. One way to gauge that is to look at their first assignment. If you feel ready to do something like that, then maybe that's a place that would give you a little bit more of the challenge you're looking for.

We have all the section preference information in and we're doing the big matching. We will email out the section assignments. Our sections mostly meet on Thursday, but some are on Wednesday and Friday. You'll get an email tomorrow that tells you about where and when your section is meeting. If you have a conflict that has arisen post the decision making, we have a little bit of an ability to do some add late and rearrangement, but it's pretty tricky. If you can make the section that you're assigned to, that's going to help us the most. We will try to accommodate you if we can to make a switch for you.

What are we going to talk about today? We're going to talk about libraries and C++ and string and stream classes. There's some [inaudible] mostly chapter three and then the handout that went out last week about general C++ has some information that's useful. There's also a library reference I gave out today which is handout ten, which is nothing but an overview that reminds you of some of the features. One other thing that's going to be the theme for the next three lectures or so is going to be there's a lot of material in C++. There are these huge libraries that do lots of things.

The goal of these weeks is not to make it so that you are an expert on all the minutia but that you are comfortable with the basic facilities and you know where to look to find out more. If you're trying to figure out how the substring operation works on a C++ screen, what are the arguments? What do they mean? What are the cases I need to be worried about? You'll know where to start and how to find that information as opposed to memorizing the whole of the library. It's not really a feasible task and not even important.

Think about it as need to know. I'm going to get you the basics so you know what's out there and then as you start to write code, you'll learn the specific things you need to solve the problem at hand. Any questions administratively? I had one student come and hang out with me on Friday. I'm hoping that was because it was so last minute. This Friday, I don't have a meeting, so put it on your calendars now for Friday hanging out.

C++ libraries – the notion of a library is really nothing more than saying you've got some functionality that you want to provide to all users of C++ or all students enrolled in CS106. There is some reasonable grouping to these things. You have a bunch of operations that operate on strings or that allow you to do graphic works or allow you to do event handling or something. The library is the packaging device in C++ where you say here's a set of routines. Typically, it comes with two pieces. One is the interface or declaration or header file. It tells you about what routines are in there – what are their names? What are the prototypes? How do you use them?

It often contains good comments about the things that you would need to know as a client using that facility – how to use it effectively and correctly. There is the code that really implements it that when you make the call to a substring operation, how does it actually work? Well, there's some code that backs it that does the operation that gets called at runtime when you make a call to that function. The libraries that we're going to see this quarter form roughly two big groups.

There is the C++ standard library, so things that come with every C++ compiler. No matter where you continue coding in C++, you will always find things like the C++ string, the C++ IO stream, the file stream which is the F stream. There's a math header. There are headers that deal with all sorts of other facilities that are beyond what we're doing here. These are the ones that we'll see most commonly in the early part are string and stream.

The typical include for them is going to be the angle brackets. That's the sign to the compiler we're looking for something from the standard header locations. One way to remind yourself about how to distinguish these from our special libraries is that you're going to see these very terse and lowercase names. Part of the legacy of C and C++ was as a professional programmer's tool, they tended to value terseness over any kind of verbose and descriptive names, making it a little easier to type a little faster to get your point across.

Things like the cout, which is the console out stream, the get line, the [inaudible] call. There's the substring name of the function there. Those tend to be short and throw away vowels where they can. They tend to be all lowercase. Whenever you're looking at a routine, you might wonder where it comes from. If it has this capitalization scheme, it's likely to be something coming out of the standard libraries.

In addition to what we have present in the standard, we also have about seven libraries that we've included as part of CS106 mostly to make our lives a little bit easier. Things like the random library or the simple IO library – they actually layer on existing functionality that is already present in the standard libraries, but the way the functionality is expressed in the standard is just a little bit awkward or unhelpful for the task we need to do. We've provided a layer that cleans it up for you.

The graphics is a good example of where there is no graphics library included in standard C++. If you're working on graphics on Windows, you have access to a different toolkit than you do on the Mac or on Linux or some other platform. We have tried to abstract out a very simple graphics library that we can run on both Mac and Windows that then we provide one interface through that in turn talks to your platform in its native language to take those Windows and drawing things happen.

Our header files are always in the double quotes. We typically use a strategy of having capitalized verbose names. We don't throw away our vowels. We try to make a little bit of sense to describe the action that's being taken. Today, what I'm going to look at is I'm going to look at one of the 106 libraries here at the beginning because it's a nice, easy one to get our head around.

I'm going to go and look through the C++ string library and then I'm going to hopefully get a chance to even start talking a little about the C++ stream library. Along the way, there will be two additional 106 libraries that help out with string and stream that provide a little bit more functionality than what's already there.

Randomness comes up in all kinds of simulations in game playing. You want the computer to simulate some random behavior – flipping a coin, rolling a die or shuffling an array or a deck of cards. Computers actually aren't capable of true randomness in the sense that you might think in the real world, but they have what's called pseudo randomness behavior where it can generate numbers in a sequence that appears effectively random from the outside even though there actually is some determinism in how it operates.

There is a set in our library – there are four functions that form the CS106 random library that are used for all kinds of random behavior that you want. I note here that they are free functions, and by free functions, I mean functions that aren't on a particular class. They're actually globally accessible. You don't call them by sending [inaudible] particular object. They just exist at the top-level name space and you can just call them anywhere and any time. When you're ready to get some random behavior, you make a call to one of these routines.

There is an initialization routine for this library. The randomized call – that's called once and exactly once in your program, usually at the very beginning to set up a new random sequence. That's what's called seeds of the generator to get it started in a new place. Then, once you've made that call, you can intersperse any number of these calls to simulate certain random events.

The standard random number generator that's in C++ provides all of this through one call. We'll generate a random number from zero to the largest number possible, and then you can decide how to map that to other things. If you want the ability to flip a coin, you want to say half the numbers are odd and half the numbers are even. You could do something like generate a number and see if it's odd or even, or see if it's from the bottom half of the range versus the top half.

What these functions do is just provide that functionality and package it up for you neatly. You can say things like random integer low to high – you say one to five. It will give you a number from one to five inclusive. If you keep calling that, you should see an unpredictable sequence of the one, two, three, four, five coming in jumbled and mixing up. There are no real guarantees about what order you'll see it that will allow you to simulate random events.

The random real – same sort of idea but in this case using boundaries that are expressed in real numbers and returning in real number. Again, the bounds are inclusive, so it can't actually return the number low or high or anything in between. The last one is simulating a probability true/false value. Given the probability of 0.5, half the time it should return true and half the time it should return false. If you give it a probability of .25, one quarter of the time it will return true and the other three quarters of the time it will return false. It allows you to simulate coin flips or other random events where you have a [inaudible] distribution.

The point of a library is to take some set of facilities that are needed, package it up, have a vision of how they work together, a naming convention and design convention that makes them coherent, that they provide some convenience and they're complete. They cover all the bases. These three provide a pretty good range of different kinds of random events. There are still other things you might need to simulate, but you can typically do it in terms of using one of these.

You could also have left one out and have to simulate the others from it, but each of them has a client use that's pretty handy, so it actually has all three of them for your use.

[Inaudible] comes out of random.h in the CS106 library. The .h is just a convention for the extension for header files. .txt gets used on text files and .cpp gets used on source files. .h is for header files, which are descriptions of routines but no real code is typically in the .h file. It's an interface file they call it.

In Java, there isn't that distinction. Everything is all in one file, but the definition of a class serves as both the description for a client using it as well as the implementer implementing it. C++ has them separated. Let me look at C++ string as the next example of something we have to make use of in getting things done. The C++ string type is actually defined in a header file, and it's a library that's added into the language.

Unlike int and bool and double that are part of the language and can't be separated from it, string is kind of an add on that's defined through a library. It models a sequence of characters including everything – letters, numbers, digits, punctuation – and the string is defined as a class. In the same way that in Java you're used to the class being the pattern from which you can declare and initialize objects that you can then message and do things with, string in the C++ world is the same sort of deal.

You have a string class. You initialize string objects. You send messages to those string objects to ask them to do things for you. Asking a string to give you the character at a particular position or the number of characters or to insert some characters or change some characters within the body of the string are all done by messaging the string. A couple simple operations – I put a little bit of string code to get started.

The variable name is actually string itself. There is something a little bit different about string when you declare it and you don't initialize it relative to the things we know about primitives. When I say string S and I don't say anything else, you might assume then similar to the primitive types that S is garbage. It has some sequence of characters. In fact, string has what's called a default constructor, one that's invoked when you don't specify otherwise such that when you initialize a string with no other explicit information, it will assume you meant to set it up to be the empty string.

Making a call string S actually declares and initializes a string with no characters. If I were to ask it for its length, which is the way we ask for the number of characters in it that's being used right here – for example, S.length and then close the open paren there. It would return zero on the empty string. We can use square brackets like the array notation you're probably familiar with to access individual characters of the string.

Applying the square brackets to S – S sub I sometimes I'll call this accesses the [inaudible] character within the string. The character's index starting from zero – so if I have a ten character string, they actually are indexed zero through nine. The C++ string – the square brackets allow you to access that character both to read it and to write it. A C++ string is mutable. The Java string is immutable. Once you create a string, it has a certain sequence of characters, and although you can make a new string and overwrite that one, you can't go in and just manipulate the string in place and change its contents.

C++ you can do that. I initialized string in this case to the string literal or string constant CS106, and then I ran a loop over the index of the proper range of indices for this string, and then I used the two upper, which is a function from the standard C library that takes a character and returns its uppercase equivalent or unchanged if it's not a letter, and then [inaudible] S of I the result of two upper.

The effect of this was for each lowercase character in the string, we overrode it with its uppercase equivalent. Any other existing uppercase or punctuation characters were left unchanged. You can make assignment into that, which is something you cannot do with the Java string.

Student: Can you insert [inaudible]?

Instructor (Julie Zelenski):I certainly can. I'm going to show you that in about two slides. There are a whole set of member functions that then do these things. This one allows you to have the sequence of five characters – what if I want to put one in the middle? I'll use something called insert. If I want to take one out in the middle, I use something called replace or erase to pull it out and put something else in.

Many of the built in operators – things like equals and less than or less than or equal to, not equal, have extended meanings that apply to strings when they're used as the operands for those types. I can assign two strings using equals. If I say string S equals T as I'm doing right here, then whatever value T is, S becomes a copy of that. S and T have the same value, but they're not related in any important way going forward. We have two copies that both happen to have the same five characters.

For example, the first thing I did after this was change the first character of T to be J, so now T is jello. S is still hello. It was initialized from the same sequence, but they don't retain any kind of aliasing from that point forward. I'm able to compare two strings directly to see whether they're lexicographically equal or less than according to ASCII ordering. I can say if S = = T – in Java, that didn't do what you wanted. It did compile, but it didn't test the thing you were hoping for. In C++, it does do what you're expecting, which is to say take two strings and say do they have the same sequence of characters.

If I have assigned S to T, if I do S = = T, it's going to say yes, they have the same five characters in the same order. Once I've changed one of them, then they'll come up as not equal. I could do less than and less than or equal to to see in ASCII ordering which one proceeds the other to do sorting of strings. Just like you think of as the integer types in double touch, those operators have reasonable meanings applied to strings.

The plus and plus equals is what's called overloaded, so extended beyond its usual meaning for addition to do concatenation of strings. I can take S and I can add to it a character space at the end, so now instead of being just hello, it's hello space. I can also add strings to strings, so I can take T and use the shorthand plus equals, which takes jello and turns it into jello jello there, attaching another one on the end.

The concatenation for the C++ string only operates on strings and characters whereas in Java, there's this kind of automatic mechanism where things like doubles and integers are converted to string and added into the concatenation. That does not happen in C++. Concatenation is just for strings and characters. If you have something that's in numeric form and you want to add it into a string, you'll have to first convert it to a string. I'll show you a routine that does that a little bit later.

I would be happy to do that. Most of the things that I'm talking about actually are in handout four was well as repeated in handout ten and in the reader. There are a million places you can look for information on strings.

Most of the heavy lifting on the strings is done via these member functions. These are part of the string class, and so these are operations that apply to string receiver objects. They're not free functions. You can't call them outside of a usage where you're saying on some receiver string, apply this function using these arguments. For example, the length member function is applied to a string.

Just to note here, the word member function is vocabulary-wise the same thing as method. Java programmers tend to call the functions that are defined as part of a class methods. C++ programmers tend to call them member functions. They really mean the same thing, but I do try to use the word member function because we are a C++ class, and that is kind of the convention. I'll probably end up using both accidentally without even noticing it. Hopefully, it won't cause you too much grief there.

The member function here is saying str.function R is saying apply the function, send the message function to this particular string with these arguments and then get its answer back or have that operation happen. I can ask a string for its length in terms of an integer. It tells you the number of characters. I can ask a string to look for a particular character or string sequence substring within the characters that that string maintains right now. It will return the index of the first occurrence found, scanning from left to right or a string::end pause. It's a little bit of a funny return value, but it is the return value that says I didn't find it.

It's a string::end pause. It's an integer value that is distinct from any other valid index within the string itself to tell you it didn't find it. Both of these have a default argument on them. We talked a little bit about that last time. If I do not specify that second argument when I'm making a find call, it will assume that you want to start looking from the beginning. If I do specify it, then it will start from that position and scan from there to the end of the string. It's a way of targeting the place you're looking for a little more precisely than just starting from the beginning and going to the end.

C++ does allow what's called overloading. In this case, the function find that finds a char and the function find that finds a string both have the same name, and so that name can be used for multiple purposes as long as there's a sequence of arguments that distinguishes them so that when I make a call to find, it knows whether the first version or the second version by virtue of whether the first argument is a character or the first

argument is a string. That can be extended to other types. This is typically used when you have an operation that really has the same behavior but some slightly different sequence of arguments is required to invoke it.

It is not something you want to use a lot to make a bunch of similar named functions that don't have similar operations. It allows for a convenience when there are two or three variations of the same theme. They might all come under the same name by virtue of overloading.

Substr is something that given a receiver string and a position in a length will extract a new substring out of the middle of the string that was received. If I take the hello string and starting from position zero take two characters, I get the string he. It copies them. It's distinct from the original, and so all it did was get its initial sequence by copying characters from there. If I go in to change the hello string into jello, that he string stays he. They're not attached in any long-term way.

Insert, replace and erase are all of the family of something that I call modifiers or mutaters that change the receiver string. You can send these messages to a string to cause new text to get added into the string, text to be removed or text to be deleted and replaced with something else. Inserting – someone asks, well, how can I put new characters in the middle? Well, I put the position where I'd like them to go. If I say position zero and I say put the string I in there, then it would bump everything down and put I in the front and replace it. If it was hello, it would be I said hello. I inserted the string I said.

The replace at a position removes length characters starting at that position and then replaces it with that character. It's a way to take a chunk out and put something else in instead. Erase does a straight remove at a position. Take this number of characters and throw them away, deleting them from the string and making it shorter. All of these change the receiver string. When you say str.insert, str.replace or str.erase, after that call, str now actually has new contents based on what you've asked it to do about changing and mutating its contents.

Here's something I should tell you a little bit about C++ string relative to Java string. C++ is kind of an industrial strength language that's targeted at professional programmers. It does not make any guarantees to you about what happens if you misuse these calls. If you give it a position that isn't valid for this string or a length that isn't valid for the string, there is no contract in the C++ libraries that said this is what will definitely happen. It doesn't say oh, it's definitely going to throw an exception or throw some sort of error. It doesn't say it's just going to truncate it at the end.

It says that the library is free to do whatever is convenient for it up to and including just crashing. It does mean that as the programmer using these calls, it is a little bit more on you to be careful that you're using them correctly and making the numbers inbounds for the string in ways that will produce correct results. It might be that it will produce a nice error message, but there are no guarantees. You wouldn't want to come to depend on that. You want to just be careful about knowing what the right numbers are.

Unlike Java, which is very attentive to those things and on your case when you're a little bit out of bounds, in the name of efficiency, it tends to just breeze through that stuff. I'm going to show you a little bit of coding together just for fun. I like to sit and show you some things. If I were to do something like want to count the occurrences of a particular character within a string, I could write a loop that looks like this.

I could say int count = zero for – and this is a very ubiquitous loop for operating over a collection – in this case, the collection being the characters in there from zero to this length. If S sub I equals the character I'm looking for, we would increment the count and then return it. I put this down here in my code and do a little testing of looking for the character C in Chihuahua cheese crackers.

Let's take a look at that and see if we manage to count the number of Cs in my list. There are four, apparently. Let's go check and see if that comes up. It looks good. We did a little bit of counting. We're feeling okay about that part. Let me do something where for example I want to remove all of the occurrences from that. I'm going to write this two different ways to highlight a little bit about how things work.

I'm going to design a remove occurrences that given a character in a string will return to you a new string where all the occurrences of CH have been removed. Easy enough. It's not going to modify the original string. It's going to return a new one. Here's my strategy. The way to build these things up is I could go through the manipulations of trying to take the characters out in place and figuring out where I'm at, but often, the easier way to do this is to build up the result – decide when to append or concatenate a character from the original string and when to ignore it and go past it.

I can do something like this where it's like if the character I've just seen is not the one that I'm trying to avoid, then I can just add it into the result. When I'm done, I have the result. If I do this and I change this call down to remove occurrences – I'm counting on the fact that result is initialized to the empty string. I didn't actually say anything there. I could, for example, do this and that doesn't change anything about it and you might feel a little better about seeing that explicit initialization, but C++ programmers are very used to seeing uninitialized strings and knowing that that means they got the default initialization to the empty string.

When I didn't find the character I was looking for, I [inaudible] the result. I'm going to switch this up just a little bit. I'm going to change remove occurrences to instead of making a new string to actually modify the string that I have. I'm going to change my code down here to match what's going to happen. I'm going to set it to do this, and then I'm going to call remove occurrences C of S, and then I'm going to print out S afterwards.

In this case, I don't expect there to be a second string created. I expect us to go in and modify that string in place, truncating some of those characters and taking them out to make this work. I could kind of do this thing where I'm walking down the string character by character and then deciding whether to collapse over it. I'm actually going to

change my strategy entirely just so I have a little practice using some of the other routines, and I'm going to end up using the string find.

We'll start with this. I can actually do this. S.find of CH, and if I don't give that second argument, it's going to start from the very beginning and look all the way through and see if it finds it. I'm going to put a hold that result at a variable and I'm going to say while found equals the result of calling find and then I'm going to stick it in. This is a very C++ way of coding. It's tightly combining this up. In this case, I have an assignment and a comparison all in the test of the Y loop.

I'm making the call to ask the string to find a particular character, storing that result in an integer here so I can use it and then comparing that resultive string end pause. So the string end pause is a little bit of a funny C++ syntax there, but the way to read that is within the string class – string:: says within the string class, scope within the string. There's a particular constant called end pause, which is used as the return value in cases like find when it's looking for something. End pause being part of the class is a way of avoiding it conflicting and interfering with any other usages where you might have variables named end pause or similar functionalities.

It's tied to the string class through the scoping mechanism. I check and see if it's not string end pause, and then if it's not, then I go into the loop here and I can do an erase of one character at position. Erase takes the position in the count and removes the number of characters I specified from that position. Then, it will come back around. I have passed string by reference coming into here, and that's a very important part of what's happening here because these calls to erase that are modifying string – if I have not passed string by reference, they'd be operating on my copy.

I'd go through all the trouble of erasing all the Cs in my copy, but when I got back out to the main call, none of those effects would have been permanent. Passing by reference really means that what remove occurrences got was access to the original S. I should make these names – I'll call this my string out here so that we don't get any confusion about the two names. The my string variable in main is really being accessed by remove occurrences without a copy. It's reaching back out into main and making changes to the my string itself.

What does it not like about that? Oh, pause. I called it found. I've achieved the same thing. That's one of the things about the string library is it's so big and has so many different ways of doing things that often two people or ten people running the same task won't even come up with the same solutions. I could have used a replace where I replaced the character it found with the empty string.

I can build it up through concatenation. I can take it down with erase and replace. I could insert the other way around. There are a bunch of things I can do that in the end will achieve the same effect but show that there are a lot of ways to accomplish the same things. The library is pretty rich and has a wide variety of tools in it.

I'm going to make one change to this to show you how I can make it slightly more efficient. This is silly because strings are typically very short, so it doesn't really matter, but I'm going to do this and I'm going to use found as my index on subsequent calls. I can say starting at found from zero, do my search from zero and then found, and then any subsequent calls will pick up where I left off. The next time around through the loop, found is at the place where I found a previous occurrence of that character and it says starting from that position now, look forward and see if you see any more from here to the end.

For a very long string like this, it ends up doing a lot less work. It doesn't start at the beginning each time. It just picks up where the previous occurrence was found and goes from there to the end. It's a small change, but no big deal.

Basically, what you're saying is find needs to return something that says I didn't find it. It could return to you zero, one, two, three, all these indices. It actually needs to return to you, and it uses a special sentinel value that says I didn't find it. You might think that might be negative one or some other thing. A good programming form would be to have a constant for it so that you don't have any magic numbers embedded in your code. That constant is defined as part of the string class. Just the syntax for accessing that constant of the string class is using the string class name :: end pause.

It's basically the syntax for I have a constant that was defined within a class. How do I get to it? I use its class name, two colons and then the name of the constant. It's just C++ for something that in Java looks a little bit more like class name dot. Question?

Student: Sometimes I see a function declaration before [inaudible] and then the definition afterwards. Is that just a matter of preference?

Instructor (Julie Zelenski):It totally is. Probably I'm being a little bit lazy in class, which is if I put the function definition up here, then I can call it down here because it's already been seen. If I put it down here, then I need a prototype up there. The prototype means I have to be a little bit more careful. When I change the name, I have to change it in both places. If I change the argument, I have to change it in both places. The problem, of course, is that when you read the code, it probably reads a little better to say here's the main which makes calls to A, B and C.

Some of it has to do with it's a little bit harder to maintain in that form, but I think it's easier when you're done to read it. You're totally free to do it either way. You should probably pick a strategy and go with it. Maintain the prototypes is not really that much work once you get used to it, and I think in the end, it probably is a little bit cleaner. When I'm being lazy in class, I'm much more likely to just throw them up there to save myself some time. It's good to note that there are a lot of things that will slip by me if I'm not being careful.

Let me go back and pick up a few last details about string that I don't want to overlook before I move away from this. There are library functions that are need to know. I have

them sketched out in a couple places, and you can look at them and see what they do – knowing they're there and then learning about them as you encounter them is a fine strategy. There are a couple additions in our [inaudible] which is a 106 specific header file which are just some things that for one reason or another are a little bit harder or more annoying to do using the standard tools than we think is worth putting on your plate for now.

We have two convert to upper and lowercase that given a string just convert it to its upper and lowercase equivalent. There are some things that do conversion between string and integer and string and real when you have it in one form and you need it in the other. Here's something that just does that for you as part of the string library. It's just some simple things that you might find yourself needing and you just want to know they're there.

Here is something that is a little bit of a bummer. Part of the legacy of C++ being built on C means that every now and then, there's a little bit of a history in our deep, dark past that pops its head up in ways that are a little bit surprising. For string, it turns out there is a little bit of a weirdness here that I want to point out before you run into it the hard way. There is a notion of the old style C string.

The original C language didn't have a string class. It actually doesn't have any object [inaudible] features at all. It did have, though, some other more primitive handling of sequences of characters. This is a very common [inaudible] to have something. I've put in parenthesis what it actually is. It's [inaudible] an alternative. Don't worry about what that phrase means. That's just for those of you who have seen it a little bit before.

That would be fine. We have this better string object that has all these fancy features, so you'd think we could just use that and ignore the fact that the other one is there. It almost – 99 percent of the time, that's exactly how it's going to work. It does turn out that there are a few situations where this old style string pops its head up and gets a little bit in our way. One way that may be a little bit of a surprise is that the string literals are actually C strings.

When you see an open quote, some characters and then a closed quote, the compiler interprets that as a C style string. It also has a mechanism by which if you tried to use it in a context where you needed a C++ string, it will automatically convert it for you. It will take the old style string and make a C++ string out of it. That means that basically I can use them wherever I want and it will mostly work out.

There is a way you can deliberately force it, if you use what looks like the type case here. This is actually calling the string constructor, and you pass a string literal or string constant. It will turn it into a C++ string manually there. It's going to turn out that you might need to know this. There's also the other problem of what if I have it in one form and I want it in the old form? I have the old form. I want a new form. I have the new form. I want an old form.

There is a member function on the string class that will return to you an old style string from a new style C++ string, and it's called the C_str [inaudible]. They let you convert. Why do you care? It turns out there's one thing you'll definitely run into, which is when you're opening a file stream, you want to say this is the file on disk that you want to identify. It turns out that that library requires the use of a C string as the name. There was a little bit of an issue trying to get all the libraries to come together at the right time, and it turns out the stream library got finalized before the string library was done, and so it depended on what was available at the time, which was the old style string.

Even years later when they're both happily debugged and working, it is still the case that when you use the stream library, you have to describe the file you want by using the old style string. If you had a C++ string variable that held the name you wanted, you'll actually have to convert it. Converting in the other direction comes up in one case, and I'm going to show you this one.

It has to do with concatenation. The plus operator that does concatenation really wants to work on C++ style strings, so if one of your operands is a C++ string, it's all fine, as long as the left or the right side is a C++ string. The other side can be a string literal, a constant, another string, a character variable – all those things work fine. As long as at least one of the operands really is a true C++ string already, you're good.

That's almost always the case. But in the case where you somehow have two things on either side of the plus, neither of which is already a C++ string – typically, that means you have a C string on one or both sides, a character on one or both sides – you are not going to get concatenation. If you try to add two C style strings, it actually won't compile.

The sad thing about these two things, about taking a string literal and adding either a character constant or a character variable is that it does compile and it just does not do what you want at all. It does so in a silent but deadly way. I'm not going to tell you what it does, but if you are curious, you can come and talk to me and I'll lay it out for you. What I want you to come away with is this memory that when I'm using concatenation to be sure that one of the two operands is a C++ string. If you have to, force one. If you have a string literal and you want it to be a C++ string, then make it one to avoid running into this.

The mistake that you get from this is actually quite mystical and very confusing. Probably 95 percent of you would never run into this, and so mostly, I've just confused you for reasons that seem unclear, but for the five percent that are going to run into this, I'm really trying to do you a favor by giving you a heads up before it causes you a lot of grief later. Just a little bit of a legacy. C and C++ go back a long way, and as a result, we sometimes have little quirks we have to deal with even in the modern world.

Student: Is that the only way you can convert a C string into a C++ string?

Instructor (Julie Zelenski):Not exactly. A lot of times, it just happens automatically is the truth. In almost all situations – if you had a routine that expected a string argument and you passed the string literal, it will automatically convert. Mostly, you won't need to do this is the truth. It happens all the time behind the scenes without any effort on your part. This is the official way to say I've got a string and I really want to force it and I'm not waiting for the compiler to do it on my behalf.

In particularly, for example, in a situation like this, it doesn't realize what you really wanted to do was convert this and then to concatenation. It does something kind of goofy based on what the old meaning of taking a C string and adding a character to it was, which was not concatenation. That's a little moment of silence for old language C that comes back to haunt us a little bit like a ghost in the attic.

Student: What do you mean by a string literal?

Instructor (Julie Zelenski):A string literal just means a string constant. It's something in quotes.

Student: Without explicitly declaring it to be a string.

Instructor (Julie Zelenski):Yeah. A string literal is when you see open double quotes, some characters, close quote, that's a string constant or a string literal. In any situation where you see exactly that – not a string variable is basically what I'm saying there.

How do we do IO? How do we do input/output in C++? Let me first say that input/output is probably one of the more distinctive features of any language. C's IO, for example, looks very different than C++'s IO, which looks kind of different from Java's IO. These are areas where for some reason, even though they all do the same things underneath it all – they let you print stuff. They let you read stuff. They have some formatting features.

For one reason or another, these are the areas where they're widely divergent in their syntax and the way you express what you want to do. That makes them particularly annoying to learn is the truth. I know a lot of IO systems, and they're all very jumbled up in my head. At any given moment if you asked me how could I print a decimal number with three digits of precision in this language, I'm going to have to go look it up. My motto is look it up. Don't worry about memorizing these details, because they are very tied to any particular language and its formatting system.

That said, we're going to use a little bit of IO. We'll need to be able to read and write things to the console to interact with the user. We're going to do a little bit of file reading, reading numbers and strings from files, maybe even producing some files. We're going to use some very simple set of features. We're not going to go too deep. When you need to know more, there are great resources to go check into for that. I wouldn't in advance go make yourself an expert on any form of IO. Figure it out when you need to.

The IOs are actually handled in C++ using stream objects. There are stream classes. The O stream is the output stream that's used for writing. The I stream are the classes used for reading. Their variance, for example – the IF stream and the OF stream are the file equivalents of the input/output streams. Cout and cin are these two basically global variables, effectively, that give you access to the console output stream and cin for the console input stream. That means the little text window that pops up that you get to type and print things for the user to see and interact with.

The standard operators for reading and writing to the stream in the default sense are the <<, which is stream insertion, and >>, which is stream extraction. You stick things onto a stream and then retrieve things back from a stream that you're reading from. A very simple example of this would be I have the variables X and Y declared here. I asked the user to enter two numbers, and then I use extraction that says from the console input stream to pull an integer out followed by another integer, and then I repeat back what they said.

In its simplest form, the kind of things you can print out are very related to things you can read in. When I ask cin to read an integer here, it looks for a sequence of digits upcoming in the stream that form a valid integer which it assembles and puts into the value X. Then it looks for another one. It typically uses white spaces as an eliminator, so any returns, tabs or spaces will be skipped over in between. Anything that led up to X, it will skip over all the white space, look for some digits and then skip over any intervening white space, and look for some more digits to pull Y.

Of course, what's likely to happen here is users are bad typists. They make mistakes that when I go to read this, what happens if they've typed the letter A or 72A45. This causes a little bit of havoc because when it goes to extract that, it looks for some digits and it finds this thing and it doesn't match its expectation. That puts the stream in what is called an error or fail state, which then requires you digging around, realizing it went into a fail state, cleaning it up and resetting and starting over.

It's not that it can't be done, but it's a little bit annoying. We just made this task a little bit less onerous by providing in the simpIO library, which is our CS106 simple IO – it has get integer, get real and get line. They all read from the console, so reading from cin, and they deal with all that error handling. They make sure that the input given was well formed. If it's not, it reprompts and has them try again. It does that until they get an integer.

When you call get integer, you know that eventually, the user will have typed in a well-formed integer and you will get that value back when you make that call. You don't have to be worried about all the machinations to check for errors. Retry is actually bundled up behind that routine for you. Most of our console input will end up using these functions just for convenience. They save us a certain amount of hassle.

I would ask – if I wanted X and Y, I would say get integer one, get integer two. I'd have to call it twice. There's not a combined form of it. It saves us a lot of trouble. I can't do it

this way. I'd have to stop it after one anyway, check to see if it failed, if not, go back in. It's kind of misleading to even show this form, because that form assumes that the user is a perfect typist and never makes mistakes, which is in this day and age not too likely.

We'll talk more about file streams on Wednesday. On your way out, look for handout five and seven for Mac or PC depending on what you're using and good luck getting your compiler set up. I'll see you on Wednesday.

[End of Audio]

Duration: 48 minutes