Week 9 Module 16 File Processing

Introduction 1.2

In this model we're going to start working with files on the file system as input into our programs. We're going to use these files to bring in data into our program and then we're also going to use files to store information after we're done with it. So this whole module is about bringing in information and sending it out to the file system for permanent storage.

File Processing 1.3

The first thing we have to understand is that there's so much data that has to be entered into most programs that it's really unreasonable to ask the user to type it in at the keyboard each time the program starts. And of course we understand this from our past experiences with Word and Excel and PowerPoint and all the other regular programs that we use generally store that information in a file. And a file is a nice compact way of keeping track of all the data that's needed for a particular run of a program. There needs to be a way to access data on the file system and in fact, we need to talk about both the input and the output. Input means bringing data from the file system into our programs and using it for processing or whatever we want to do with it. Output means taking the data from our program from main memory or from wherever we have it stored in variables or vectors or arrays or wherever, and putting it out on the file system. And this is actually a two part process; there's actually two steps here. So that when we're working on a file we would have read that file initially, we would work on it in main memory, and then we would output that file back to the file system.

Files and Locks 1.4

Files all contain a name and we know that. But what you might not realize is that files also contain what's called an extension. And the extension tells the operating system whether it's Windows or MacOS or Linux or you, what might be inside that file and what program might be useful to look at the information in that file. So I'm sure your experience with Doc, DOC, which is often opened in Microsoft Word, or DocX which certainly is open in Microsoft Word. You might also be familiar with TXT which is a plain text file that contains no formatting, it contains no extra information; it's simply text. There might be an XLS file. And all these file extensions serve the purpose of telling us what program we want to use to open that file.

So to open the file, you have to know its complete name. So we're not going to have a nice graphical user interface for our programs, at least not initially, that allows us to select the file. Instead we're going to ask you or the user at the console, to specify the file name; and the filename includes the extension. So whereas in Windows you might look at the files and see file without an extension, in reality there's probably an extension that Windows is hiding for you. So the real filename would be something like file dot TXT.

In addition we know that the file system is organized in directories and you might know these as folders on your operating system. In reality it's simply a list of files that are contained inside that directory. And for simple purposes at least for now until we get into complex operating systems concerns, the file exists in just one directory. And a directory can exist inside of another directory and so on and so forth; we call this a hierarchical database, which we'll talk about later. If you don't specify a directory when you're opening a file, then the operating system assumes that you mean the current directory.

Now the question then becomes what is the current directory. And if we're working in Visual Studio and we try and open up a file then the current directory is wherever we have stored our dot CPP file. If we're running an executable program then the current directory is most likely where that executable program is. So this brings up a little bit of diversity. If we run the program inside Visual Studio, it's going to look in the current directory for any files you want to open and that current directory is the dot CPP file. But if we then produce the EXE the executable program and run it manually through windows explorer, it's going to be looking in a different directory because the EXE and the CPP files are not in the same place. So you kind of have to keep track of where you expect your programs to look for these files and you have to put them in the right place. Now of course if you don't want to do that, you can specify another directory at least on Windows you can specify using a backslash. So you can start at the top of your C drive and do backslash users backslash your name backslash so on and so forth and so on and so forth and you can specify the entire full path name for that file and then it will open appropriately. Most of the time we're not going to be worried about doing that so we're going to be looking at the current directory. And this holds true for Windows it holds true for MacOS, Linux whatever you're using.

Now the other problem about this is that files can be locked. So if a file is in use by one program, it can't be used by another program at the same time; and this is to protect the consistency of the data inside that file. Now the only exception to that rule is that if both programs want to read the file as input, both programs can have access to that file. So we have what's called a locking system: there's a lock to prohibit reading of a file that's being written to, and that's really what it amounts to.

Objects 1.5

So C++ has an internal representation object for the files. So there's an object inside C++ which represents our connection to the file. The data type of that is going to differ based on what we're going to do with it; whether it's an input file or an output file. And if we want to have an input file, then the object type that we have to create is called an ifstream. And if we want to have an output file, then the object type we have to create is an ofstream. So it's fairly easy to remember these as fstream objects, and the i is an input file stream and the o is an output file stream so ifstream or ofstream. You have to create the appropriate type and you can't change what you're going to do with that file once it's created. But of course you could always close it and open it is a different type.

Steps to Creating 1.6

To get this to work, first you have to include fstream. So we put at the top of any program that we're going to use files we're going to put pounding include fstream as you have seen it here. Once we've done that, we have the ability to create those objects of type ifstream and ofstream which is appropriate to the action that we're going to perform. And then we have to open the connection to the file in the file system. So it's not enough to just create the object, we have to actually connect that object to the file in the file system. Now be very cautious, because opening a file could fail. There's a lot of situations which could cause either an output or an input file to fail so it's really important that we check that with an if statement after we've made the attempt to open the file. Once the files open the object can be used for import now put in really the same way that we've been using the cin and the cout variables. We can use the output operator and the input operator appropriately for whatever the data type is we're working with, whether we're doing input or output. And in reality it works very much the same way.

Passing to a Function 1.7

When we're going to pass an ifstream or an ofstream an object to a function, or return it from a function as we'll see later on. Those have to be passed or returned by reference. So the ampersand has to be there. The reason for that is that the act of writing to or reading from a file, actually changes the object. Specifically, we have inside the file object, inside the representation, inside C++, we have a pointer to how far into the file we've been working with. So that when we read in, we're always reading in from where we left off on the last read in operation. Now obviously any read in or write out operation is going to change that file pointer, and that's an act of changing the object which means it has to be passed in by reference or returned by reference from a function.

Cin and Cout 1.8

Cin and Cout are also objects. We've been working with them over the past few weeks and we didn't realize that they are actually also objects and their data type is an ifstream and an ofstream. Now since we understand inheritance, we know that an ifstream object is actually an istream object as it's base class. Every ifstream can be treated as if it were an istream, and every ofstream can be treated as if it were an ostream object or said another way, if we design a function to accept istream and ostream by reference of course objects then we can also accept ifstream and ofstream objects. Which is a really nice convenient way of saying that we can create a function that can accept either a file or it can accept the screen or keyboard for input and output. And that's very helpful for debugging purposes if we create a function to do all the processing, to do all the output or all the input, and then allow for debugging to send either Cout or Cin to that function.

Output 1.9

So C++ makes output relatively easy and it's not really very difficult. All we have to do is create one of these ofstream objects; that's the first step. So we create the ofstream object and of course we give it a variable name like out file is descriptive enough to tell us what that file is going to be, and then we have to open it. And when we open it we tell it the full filename that we want to open as you see here we've got filename dot TXT. Or you could also if you wanted to skip the step of opening it, you could also just use the constructor for the ofstream class and pass the filename to the constructor. Only in very rare instances will opening an output file fail; but it still could. For example, you might not have permissions on the drive to create a file, or the drive might simply be out of space. If any of those things happen, we're going to see how to detect that opening a file has failed in just a few minutes. But you should check for that every. If it's important that the file is going to open, you should check to make sure that the file does open. Now once the files open, you can write to the output file exactly as you would write to the screen. So it's a simple output operation you can say outfile arrow arrow, or output operator. And then print out your string or print out your variable or any capability that you would have to write it to the screen output to see out, you have the ability to write it to this output file. And it works again very much like printing to the screen.

More on Output 1.10

So what happens when we actually open up the file? Well the first thing is that the program, your program, will ask the operating system to open the file. And what the operating system does, for output files, is it checks to see if the file exists. If the file does exist, it's going to be erased and a new file created in its place; so you'll lose the contents of that file. If the file doesn't exist then it's created and either way your program is going to receive back the connection to that file, so that then you can start to output to it. It's really important that when you're finished working with a file, that you close the file.

And there's a function inside the ofstream object called close. So you would just simply say out file dot close, for example, and that would close the connection. Because when you're writing to a file what you have to realize is that, you’re actually sending information into a buffer; it's into a memory space. And the operating system is then told a periodic interval to take the memory out of that memory space and physically put it on the on the hard drive.

What happens if your program crashes or the system shuts down when your program has already written to that memory and hasn't yet flushed that buffer out to the hard drive; you lose all that information, it's all completely lost. So it's best to make sure that you close the file. The other problem of course is that if we have an output file we're doing writing to this and, we remember on the discussion on locks just a minute ago, that if we're out putting information to a file we've locked that file for reading. So if you still have a connection to the file if you still have the file object open, nobody can read in and that includes you if your program continues and tries to read in the data that it just wrote out. It won't be able to because of that locking mechanism. So it's really important you close that file connection when you're done with it. If the of stream object is destroyed because it falls out of scope or if your program ends, of course the file connection is closed in the buffers flush so everything's written at that point. So you don't have to worry about it if your program ends. But if you're going to read in later on it's really important that you close that file and reopen it.

Example: Creating a File and Outputting Information 1.11

I just wanted to start by talking a little bit about what we can do with outputting a file. So we'll go through the quick procedures of how to create an output file and where that information is stored and then we'll talk about putting some information in the file. So of course as we discussed the first thing that's important is that we pound include fstream. Okay. And from that point we then have options of creating an ofstream object and I'll just call my object, out file here. When I do that I get an object of type a left stream which I can write to and it's not at this point connected to anything. So we haven't specified a file name or anything like that at this point, so we don't really have a file. In order to create or rather connect this object to a physical file, we can do an open operation and we can just give it like file name or let's call it out file dot TXT. And when that happens the file actually gets created. Now if the file already exists it's going to be overwritten, we know that, and if the file doesn't exist it's going to be created. So I'm just going to compile this and give it a run, just that we see.

Now the question becomes where are these files created because we haven't specified where the file is supposed to be created. So what it's going to do is create it in what's called the current directory, and for visual studio the current directory actually is where the CPP file is. So I'm going to go ahead and open that here; you can see I hope the folder view here and I've got this CPP file and in the CPP file I've got out file dot TXT. So I've got the output file TXT; of course, it's zero bytes because we have to put any information into it. So it was created here and you can see the creation times 12:37 pm, if I run it again I get a different creation time. So now if I go back I'm going to see is 12:38 because now a minute later. So we are creating those files successfully. We also have the option of doing it in one shot of just simply creating the output file and calling the open using the constructor. So I could have done this; although I'll comment it out. I could have done that because you'll see the combination. That would make it not necessary to call the open operation on that file; it'll be opened. How do we output thankfully we can use the outfile object very much like we've been using Cout since the beginning of the semester, and here I have just a “hello world” and I'm going to run that program. Now you're going to see that there is some information here in this output file a file dot txt and if we open it up you're see “hello world.”

Input 1.12

Just like the ofstream object, we have an ifstream object which we can use to read information. And we can use the dot open or we can use the constructor for the object to open the file. Unfortunately, input files are a lot more likely to fail when they're opening and the reason for that is more than likely somebody inputted a bad filename. Remember that we're not selecting the file graphically for through a graphical user interface, we're writing in the filename. So if you miss typed the file name, what should the program do. Well when you try and open the file using the ifstream dot open or using the constructor, if the file doesn't exist opening that file will fail. And we won't get a message back from C++; we have to check actively on whether that succeeded or did not succeed.

So there's really no response from the dot open function, we just make the attempt to open the file and then it's our responsibility to use an IF statement to check to see if opening that file succeeded. Now we can use a simple bool member function inside the ifstream class which allows us to simply say if in file and that's enough to check to see if the file is open and connected properly or if it rather if the ifstream object is open and connected to the file properly; but if we attempt to open it once and it fails then who's to say that the second time we try to open it will succeed. So the problem of course with an if statement is that we're only going to check it once, it's actually better to use a while statement and put all the code for opening the file connection inside the while statement because if the while statement doesn't succeed we won't continue on. So doing while in file, as you'll see in just a minute, is actually a better solution because it allows us to prohibit continue continuing the program until the file actually is opened successfully; whereas, if will only try once and then whether it succeeds or doesn't succeed it's going to continue on. If you are going to try opening the file again as in a while statement or an if statement and not just fail out and say we've got a problem, then you have to understand how the flags operate inside the if stream object.

When we try to open that file and the file openings fails then a flag is set up inside the stream object to tell us that something has failed. Now unfortunately with the ifstream object even if we have a successful opening later, the flag to tell us that a failure has occurred will still be set. So the solution to this is to do a clear inside the stream object; there's a function called Dot clear and when we before we attempt the opening again we're going to call dot clear. So to just walk you through the process, we'll try to attempt to open the file once. If it fails we'll enter into the While statement to say that it has failed. Then we can check to see that the file has failed to open and then we can call clear to clear the flags and open to open it again. That way if the file succeeds in opening the second time than the flags will now be clear appropriately so. So we have this multistep process towards checking how to open a file, and then once that process is complete we know for sure that the file is actually open.

Example: Opening a File Stream Object 1.13

Now I'd like to show you how to do some input from a file. And so what I've done is create a couple of sample files here; I'll just show you the first input file. And again it's very important these are text-based files, so if you're creating these files on a Mac you have to make sure that these are created as regular text. If you're creating them on a PC you might just use notepad; notepad is really easy to do to create these files and you can just go and right-click inside the folder and choose new text documents. Don't choose word or anything like that because if you look at a word file it actually contains quite a bit more formatting information and that sort of stuff. We're not going to really care about any formatting; we're just working with text.

So here I've created a text document with just a number of integers and what I'd like to do is create a program, which can read in and maybe find the average of these integers. So we'll keep it very, very simple to start off with. So the first thing is of course that account include my fstream again. I'm going to give you the open input file function and open input file is going to have to take into account the fact that the user might type in the wrong file name. So it's going to pass in the ifstream object which an ifstream object we talked about is a file coming in and when we pass streams to functions, of course we always pass them by reference. So I'm going to pass in my in file, by reference, to my open input file function and I'll just create a string for the file name here and ask the user the file name and I'll read in its file name. And then I'll go ahead and open that file so I'll assume that the in file objects not opened at this point which makes sense since that's what we're asking this function to do. And we'll give it the file name I haven't included string here so I can see that these strings.

We hope that the file name opens but hope is not enough we're gonna have to verify that the file actually does open. So I'm going to write a while loop to see if the file is open and you ask why it's a while loop, well because what happens if we try again and it still fails to open a second time and a third time and a fourth time. In fact, you could ask the question how many times is it necessary to try to open a file before successful and the answer is we don't know; we want to continue in this loop till the file actually opens. While in file is not open so while a file is not successful while we have a failure on any file it can say file fails to open and then we can ask the user to do the same process over again: read in in that file name, and then we can try and open it again, but before we open it we have to call clear because what we're testing for is a flag that shows failure. And even if we have a success the previous fail will cause that flag to remain, unless we've called clear so before we open it again before we attempt to open this file again we're going to have to clear it that failure flag and then you can try and open it on the file.

So it's not exactly the same code as previous because you have to clear the flags but we do want to try opening it again and in order to test it right after that. We're going to have to make sure that it's not in a failed state to begin with and then we can try opening it and see if it's a failed state. This function almost becomes wrote because you can use it over and over and over again for different projects that you have to work on, so I'd commit this to memory or at the very least get used to knowing how this function works. And that way when we create an ifstring object called in file we can use the open input file function to open that in file and trust that when it's done that that input file is open. So now we know that in order to get out of that function the input file must be open and ready to read. From that point we can read in maybe into an array or a vector the elements of the file, so I know that these are integers.

I'm going to go ahead and read these into a vector of integers, so of course I'll pound vector and then I can have a temporary one so I want to make sure that I can't read in into V there's no simple way to do that. But what I can do is create a temporary object and read in in a while loop from in file into the temporary object; now, what this does is reads in one integer into temp and then we can process it. Now if the input operation if the read in operation is a temp doesn't succeed then we're not going to continue in the while loop, that would be a failure and we'd fall out of the while loop. So like when the file is done when we've read in all the information from the file then the read in operation will fail and we can go on and do something else, but as long as we're able to read in an integer then we can store it in temp and then do something with that. And for this we're going to be real simple, we'll just push that back the push temp back out to V and then if we want let's say the average we can take the sum using our for loop, V sum plus equals the average of the integers we can do some divided by dot size, so that tells us a quick average of the items in the file. And I've called this input one dot txt very important in

Windows that you show the file extensions, so if you don't show the file extensions this is only going to show up as input one but it's very important that you recognize the file extensions do actually exist. So if all you're seeing is input one and it says text document then in reality this is input one dot txt. I have it showing all the file extensions for all my files because that's the way I like it.

I'm going to compile this program I didn't initialize it. Yeah, that's right okay. So I've got to initialize that and then when I run the program txt. The average of the integers in the file is 55 so that's an easy way to read in a whole slew of integers there and you can work on those integers using a vector or a dynamic array of some sort. I'll show you the code just one more time main and finish the open it

Reading in Data 1.14

So how do we, now that we have the file connection open, how do we now read in from that I asked stream object. We've been doing it the same way with as with the keyboard. The only difference here is that in the keyboard instance, we had to wait for the user to input the information so that we could read it in into our program; now everything is open and everything is available inside the file, so we don't have to actually wait for anything to happen. All the read in operations can happen before anything proceeds we can do the entire read in of the file before we proceed. Now one of the questions is how do we detect when we've run out of stuff in the file or even better question is how do we detect if the file never contained anything to begin with; maybe it's just simply empty. And you'll see a lot of books and a lot of it's information on the internet about using dot E.O.F. but I want to caution you against using that because a lot of a lot of these instances don't understand what E.O.F. does: E.O.F. tells us that the end of file has been reached. But unfortunately recognizing that the end of file has been reached requires that we read in that end of file marker. So if we open for example a completely empty file then E.O.F. will not have been reached because we haven't read in the end of file marker, and if we start to then read in and process information will be processing information that doesn't really exist; we will have garbage information.

So I caution you against using E.O.F. and the better solution to that is to do a while infile arrow arrow O. temp, and what that does is read in the file read in a piece of information into that temporary variable, whatever the data type is not really important. The first piece of information in the file will be read in into that temporary variable and then because we've contain this inside of a while loop the while loop will test in a infile in very much the same way that we just tested it to make sure that it was open successfully. So it's a two-step process, which does both the read in operation and the test to see that it was successful. So we've got a nice compact way of both reading in and testing that it was successful at the same time through this simple while loop format; which works out a lot better than ever then E.O.F. ever actually did. The input operator, we have to understand how that really works, and we might have gone over it in the past but it really bears repeating here. The input operator in C++ is going to skip over all leading whitespace characters; whitespace characters are r your space your tab your return characters anything which does formatting inside the file but we don't actually see.

So, if you were open this in Notepad it shows up white but it's really there. It skips over all the leading whitespace characters and then reads in any valid characters and we'll talk about what valid characters are in just a minute and then it stops when it reaches any trailing whitespace or any invalid characters. So that's the three step process that the input operator is going to doL its going can skip leading whitespace read invalid characters and stop when it reaches trailing whitespace. And if we put this in a loop and constantly read in, then what we're going to be doing is skipping any leading whitespace characters reading in the valid characters and then stopping at the trailing whitespace character but then the next read in operation will skip over those whitespace characters. And if we've done this correctly we can read in the entire file and when we get to the end, we will have completed everything and everything will be working exactly as we expected. So, that's what we're going to aim to do in reading in data.

What’s Valid 1.15

So we understand that now we're going to do this three step process of skipping leading whitespace characters, reading invalid characters, and stopping when we reach trailing whitespace or an invalid character and the input operator is going to do that for you automatically; We just need to know and understand what's happening internally inside that operator. So what constitutes a valid character really depends on the data type that we're trying to read into. Remember the operation is something like in file our temp. So what's the data type of temp and that really defines what valid characters are. Now temp is a string then anything goes any characters a valid character and we're going to read in a word at a time if you will, so effectively if we had a somebodies name on a line it would read in something like Daniel and then stop at the space between the first in the last name. And then the second read in operation would read the last name like Katz then that's my name. So we need two read in operations to read that in; that's for strings. Now what about integers; we can't read in letters into an integer so a letter would be constituting an invalid character for an integer. Integers can only accept whole numbers. There's no period a period is an invalid character if we're expecting to read in is an integer. But for a double, a period is a valid character but if we have two periods that second period constitutes an invalid character. And for a character, a char the first character is valid but a second character is invalid.
So whatever the data type is C++, will process the data type appropriately, will process the input appropriate for that data type and its skips over only the invalid characters rather it's stops when it reads the invalid characters.

Getline 1.16

So, we know how to read in one individual item in the individual string, an individual int or an individual double, but what happens if you want to get a whole line of text and process that whole line of text accordingly. So, in my previous example of reading in a name, what happens if the person… Some people have a name like Daniel Katz and some people have a name like John J. Jones that would be two items to read in for one person and three items to read in for another person.

We can use the getLine function to get everything up to the end of the line and that's a really useful function. It doesn't skip leading whitespace so this doesn't work the way that input operator does; it doesn't skip any leading whitespace. It captures everything in the line so all the characters in the line. And so we reach the return character at the end of that line and then it stops, and it does not return the return character. I know that's a little bit of a strange way of saying it but the return character at the end of the line is removed from the stream and we don't return that into the string. So the way to call this is just getLine and we have inFile and myString and what will happen is from the in files current pointers, wherever the current position inside in file is, will read in everything up to and in the up to and not including the return character at the end but the return character at the end would be removed. So if we put this in a loop, for example, we could get an array of strings and each of the strings would not contain a return character and it would not really skip the leading whitespace it would include everything on the line.

So, we have a really nice way of reading entire lines at a time and then processing them in memory. The only thing to take note of is that if the file pointer is pointing to a return character then getLine is going to return nothing because its gonna read a no characters and then throw away the return character also. But this is a great way to get a large chunk of data out of the file when you really don't know how many whitespace characters there are, you can just get the whole line or the rest of the line and then you're set up to read in on the very next line.

Ignore 1.17

So we can always use the ignore function if we want to skip over a set of characters. Now what the ignore function does is it allows us to tell it how many characters we want to skip or what character we want to stop at skipping. so you'll commonly see something like in File dot ignore, and then nine nine nine nine and then backslash n in a character in a single character scene and single quotes there. which means we're going to skip over nine thousand nine hundred ninety nine characters or the first return character that we see. so it basically skips to the end of the line as long as the line is not more than one thousand nine hundred ninety nine characters. and what this is useful for is if we want to read in part of the line, not the whole line, and then we want to skip to the end of the line. we can use the read in operation like the input operator and then we can once we've got the information that we want we just ignore the rest of the line and then go back to using the input operator to read in the next lines piece of information that we want.

So the ignore function is actually a very useful function. Sometimes we might be at the end of the line, because we just did an input operator and we stop the trailing whitespace, the return characters trailing whitespace, and the next line we want to read in the whole line. if all we did was a get line we would get nothing because we were really looking at the return character, but we can first do it an ignore and skip over that return character and then read in using the Get line then the whole next line.
So get line and ignore kind of go together and ignore is the solution to having to do get line after we've done an input operation using the input operator. we put in ignore in the middle to make sure that we skip over that return character and the next line we're going to get the whole line As opposed to a blank which is nothing at the end of the line.

Example: Opening a File Stream Object 1.18

Alright, now I'd like to show what we can do with something a little bit more complex; maybe for an input file. We can take a look at this input file and it has some student IDs and test scores and names associated with each other and tab characters in there. And, let's imagine that we were given this file and you can imagine it's a much bigger file than it is, but we have guaranteed in column one student IDs and in column two we have student test scores and in column three we have the students name. And what we'd like to do maybe is order these by test score or we like to find out all the students who had a test score higher than 90. So, lets load this data into a vector inside C++, and then we can start to work on it. So just keep that format in mind: student ID on the first column, test score on the second column, and student name on the third column. And what I'm gonna do is create an object; so, now we have the ability to create an object and I'll do this easily, I'll just do it as a struct. And here we go: Student and we'll say that the integer; we have an ID that's an integer we have test score and we have a string that's the students name.

So that's the format of the file and I can create a vector of students, call that VS. S now I've got a vector of students and I'm gonna create one student, again, so this is going to be my temporary student. Now when I go to read in, I've already opened the input file using my open input file function that you saw previously. And I'll go ahead and read in into the temp dot ID, and I know that I'm going to have to do that inside of a while loop because this is going to happen over and over and over again. Now once I've done that, I've guaranteed that I've got a student. And I know that some of you might ask the question of what happens if the file is corrupt; my answer is that if the file is corrupt you really can't do anything. So we're assuming that the file is complete and that on every line we have a student ID, a test score, and a student's name. So, if we read in into the ID, we then still have the additional problem of reading in the student's test score and then we have the problem of the name.

So, now I've filled in the temp ID and I've filled in the temp test score, but the problem is going to be that this name contains multiple breaks. Now we remember that in a read-in operation, we read in valid characters and we stop when we reach trailing white space, like tab character, or we reach an invalid character and we're gonna say that we're not gonna reach any invalid characters anywhere on this file. But how do we go about reading in an entire name? So we've read in, let's say, the first student ID, and we've read in the first test score. And the file pointer is now sitting at this point because it stopped when it reached trailing white space, and what we'd like to do is read in this. And that looks simple; it looks like two names. The only problem is that if you look carefully at some of these lines, it's not two names, its three or perhaps even more names. But the problem is that we're going to have to read in basically the rest of the line. So, what I want to do is go ahead and read in the whole line, starting from the point where we left of, which was the end of the test score and going to the end of that line. And I have a function for doing that; it's called getLine. And I can give it the in file and I can give it temp dot name and it will read in the rest of the line.

Now I know that getLine seems to describe that it gets the entire line but in fact getLine begins reading from the point where you left off. And that might include that tab character there; in fact, it probably will include that tab character. So what we might do instead is before we do that we can either get the one character, which is the tab character, or what we can do is that if we know they're tab characters, we can tell it to skip the tab character by saying inFile dot ignore. And inFile dot ignore needs first a number of characters that we're gonna skip and we know this is just going to be one character, but we'll put in nine thousand nine hundred and ninety-nine, just for the sake of putting it in. And it'll stop when it reaches that first character that it should ignore. So, it'll stop either at ten thousand characters, nine thousand nine hundred ninety-nine characters, or it'll stop when it reaches the first tab character. And what that'll do is that it'll skip over this tab character and then getLine will read in the rest of the line. Of course, I've only read into temp and now it's necessary to go in and push back everything onto the vector.

So I'm gonna have push back temp onto the vector, and when I finish with that I should be able to write a simple for loop that gets the student and we can test to see if S test score, we said greater than 90, and if it is then we can output the students name. I'm gonna go ahead and put in a little description here, and let's go ahead and run that. Okay. Looks good. Input two dot TXT. And you can see that students with test scores over 90 are Daniel Katz, George Washington, and E. F. Johnson. And if we take a look, that is correct; it looks correct. We did not have John H. Jones because John H. Jones was not over 90, he was 90. But that's perfect; that's exactly what we expected. So, what we're doing is we're reading in the individual items; so, we're saying we've seen an ID, once we've seen the id we know there will be a test score, and we know that there will be a tab character, and we know that there will be a name. And we don't know how many items there are in the name but we'll go ahead and grab that and push it back onto the vector and then once it's in the vector we can start working with it. So that's an easy way to start working with a lot of files; files with a lot of data in them. If you don't need the getLine, you can just read in straight into temp ID, temp dot test score, and stuff like that. You can keep read in operations, you can tie the read in operations together. So, you might do something a little bit more insightful like that; that would work as well. Okay, you don’t have to but you could; I like to do it just this way but whatever works for you, works. So that’s the way we can read in a large amount of input information and begin processing it.

Seekg 1.19

There's another function inside the stream object which is useful, called seekG. And seekG allows us to move the file pointer around skipping over characters either forwards or backwards. We can also use seekG to move back to the beginning of the file. So, if we specify a positive number that moves us that number of characters ahead in the file. If we specify a negative number that takes us back that number of characters, but if we specify as zero takes us back to the beginning of the file. Now one of the things to take care of here it with seekG is that if we've reached the E.O.F. marker, if we’ve reached the end of file and the file is now what we consider in a failed state because we read in the end of file marker, using seekG to send it back to zero doesn't clear those flags. So if what your point is, is get once you're at the end of the file is to move it back to the beginning of the file, make sure you call it clear because it's going to clear those flags also.

Reading and Writing 1.20

So, what happens if you want to do both reading and writing, and I mentioned this before but it's bears repeating. Surprisingly, it's actually strange to want to do both reading and writing at the same time. and I know the concept that you have in mind is one of like well what if I open up a document in word and I want to edit that document. Well, what word actually does is read in the entire file; it’ll read in your Word document into main memory and when you make any changes you're not making them in the file, you're making them in main memory. And anybody who's forgotten to save the file knows exactly what I'm talking about because when you go back if you hadn't saved it, it won't remember any of those changes that you made. So what's happening is we're reading in the file making the changes in memory and we're writing back over the same file. Which is useful because if we want to save the information out we can either save it out to the same file or we can save it out to a different file altogether. Now usually we'll read in that entire file, will make those changes in memory and then we'll write the file out to the disk and that's a normal process to do; that means we're not going to be doing both reading and writing at the same time. So what you would see is the opening of an input file, the complete reading in operation of the input file, and then calling close on that input file. Then we would do whatever changes are as necessary, open the output file with the same file name, write everything out and then close the output file. So that's really the process for doing both reading and writing; in other words, in place modification of the files is not a normal thing to do.

Appending 1.21

One thing that we may want to do, which is actually very common to do inside programming and inside the file system, is adding data to the end of a file. So you might have seen a log file, you might have a log file which keeps records of transactions or you might just want to record a large amount of information and you don't want to have to read in the data and then add to the end of it and write it, out as we've said we would have to do. So there is an option inside C++ called appending and what appending does is leave the contents of the file alone. So in the original file, nothing will have changed and what we do is add on to the end of the file.

To do this we have a second parameter that's passed to the open function so when we open it, we're going to specify the file name first and then we're going to specify this IOS double colon APP. And that's the way to tell C++ any ofstream object that we don't know that we want to append to the file and not overwrite it. If the file doesn't exist, it’s going to be created and then we’ll add on to the end of this brand new blank file, so it's effectively like writing. If the file doesn't exist, then IOS APP really has no impact on what happens internally in the ofstream object. However if the file does exist, it sort of does seekG, if you will, to the very end of the file and then any information that you write into the file is going to be added on the end of the file, rather than overwriting what's existing there. So, if what we're doing is taking in data from the user and storing it on the file system; we might take in some data in one run of the program, write it out to a new file and then the second run of the program, we don't want to overwrite what the user already gave us, so we'll just append on to the end of the file.

Review 1.22

In this module we saw how to open files for both reading in and writing out. We saw how to write out information to a file; how to read in information from a file. We talked about some of the pitfalls of opening files as well as reading in and writing out. We saw how the input operator works, specifically the three-step process of skipping leading whitespace and then reading invalid characters and then stopping when it reaches trailing whitespace. We saw the use of the getLine function, the ignore function, we saw a seekG, and we saw how we can append onto the end of the file. So at this point you should have the abilities to write into your code file processing routines so that you can bring in information directly from a file. And we can now you use very large datasets, which allow us to do quite a bit of information processing, and we don't have to worry about a user sitting there for hours typing in on the keyboard.