University of Illinois at Urbana-Champaign
Department of Computer Science

MP5: Trees – Word Counting, Part 1

CS 225 Data Structures
Spring Semester, 1999

Handed out: Thursday, March 4, 1999
Due date: Wednesday, March 10, 1999 at 11:59 PM

1 Introduction

In this MP, you will gain a bit of practice with binary search trees by using one to count words in a file. You will also get a bit of practice with C++ streams – specifically, file streams, as you will be writing to and reading from a file. The "MP spec" part of this handout isn't really much more than the README file that was posted earlier, since the class you are writing really is quite small. However, since you have not had much experience with streams yet, the majority of this handout is a discussion of C++ streams and file I/O. This should make understanding the given code and what it does somewhat easier, though with your C++ reference book and a careful examination of the given code, you could accomplish the same thing yourself if you were looking to make life harder on yourself. :-)

2 The programming assignment

To begin with, you will need to copy the given files into your own directory. The files are located in:

```
~cs225/src/mp5        (MP code)
~cs225/src/mp5/test   (test files)
```

3 C++ streams

Streams are the tool used for all kinds of I/O in C++. A stream can be defined as a sequence of characters (i.e. a sequence of bytes), and the various I/O operations simply direct these streams to various places, so that the bytes flow from, say, memory to the screen, or from a file to memory, or whatever else you are trying to do. Conceptually, data enters the stream at one end and exits the stream at the other. For example, when you use a statement such as:

```
cout << x;
```

the variable `cout` is the *standard output stream*. Conceptually, one end of the stream is attached to memory, and the other end is attached to the screen, almost as if you had a garden hose running from memory to the screen. You then enter your data into one end of the stream, the memory end. The data travels through the stream just like water travels through a hose, and eventually the data flows out of the other end of the stream, the monitor end, and onto the monitor for you

to see. Now, this is not necessarily the most precise description of the low-level details, but if you keep this conceptual picture in your mind, then it will help as we deal with streams.

Generally, the only part of the stream you have control over is the memory end. For example, when you use an output stream such as `cout`, the ultimate output end is already attached to a device of some kind. The output stream `cout` is attached to the monitor. Another output stream (let's call it `fileOutputStream` just to give it a name for now), might have its ultimate output end attached to a file, so that you can write to the file. And, you might have an input stream called `fileInputStream` that has the initial input end attached to the file, and the output end attached to memory, so that you can take information from the file, pour it into the stream, and have it pour out of the stream right in the middle of your memory, so that you can take this data pouring out of the stream and store it in your variables.

At the non-memory end, the system knows how to deal with the data using a form appropriate for that device. That is, as the data pours out of `cout` at the monitor end, the system knows how to take that data and present it in the appropriate manner on the monitor. As the data pours out of `fileOutputStream` at the file end, the system knows how to take that data and actually write it to the physical implementation of the file. When you want data to be poured into `fileInputStream` from the file you are reading, the system knows how to access that physical file implementation and get at the data so that that data can be poured into the stream. So, generally the non-memory end is already handled by the system, and you are concerned only with getting data from memory into the output stream, or from the input stream into memory.

In addition, the memory end of things is handled for the basic, built-in C++ types. If you are placing an integer into an output stream, the system knows how to do this. This is why code similar to what you have already seen, such as:

```
int x;
x = 5;
cout << x;
```

will work. You are trying to place an integer into the output stream, and the system knows how to do that.

So, in general your only task is to add stream capabilities to your user-defined classes. Since all user-defined classes are ultimately made up of built-in types (your class may be something like "a user-defined class of user-defined-type member data, each of which contains user-defined-type member data, each of which contains built-in-type member data", but eventually at the lowest level the classes are made up of built-in-type data), and the system already has instructions for stream input and output of the built-in types, you are guaranteed that you can write any object you want to a stream, or read any object you want from a stream. The format you use is up to you, but since all objects are basically collections of built-in-type data, you can write any object to a stream simply by writing the individual member data to a stream, and you can read any object from a stream simply by reading the member data from the stream one by one and then putting together the object. These tasks will generally require private access to the object, however (so that you can read or write the private data), so the function performing the stream access will be a `friend` of your user-defined class so that it has that private access to your class.

4 Adding stream capabilities to your classes

Behind the scenes, in the actual language implementation, there is an entire hierarchy of classes that work together to support I/O. Two of them are `istream` and `ostream`. The class `istream` is a

general input stream class, and the class `ostream` is a general output stream class. Whenever you have written your MPs, you have included the file `iostream.h`. It turns out that this file simply includes the headers for `istream` and `ostream`, so that you have them in your program as well, and otherwise all it does is to create the objects `cin`, `cout`, `cerr`, and `clog`, which are all stream objects. The variable `cin` is an object of type `istream`, and the other three are variables of type `ostream`. So, `cout`, which you have already used extensively, is an object of type `ostream`, i.e. it is an output stream object. Likewise, `cin` is an input stream object. (Never mind the other two for now.)

Now, you have used `cout` to print values to the screen as follows:

```
int x;
x = 5;
cout << x;
```

What is really going on here is that an `ostream` member function is being called. The following code will actually compile and run:

```
int x;
x = 5;
cout.operator<<(x);
```

There is an `ostream` member function:

```
ostream& operator<<(int n);
```

that is being called above. There is a similar function for every basic type in the language; the function `operator<<` is simply overloaded many times, each time with a different parameter. So, no matter which built-in type you try to write to an `ostream` object, it will work because the class `ostream` supports those operations. The compiler simply figures out the correct member function to call at compile-time, based on the function headers that are available and the type of the variable you are trying to write to the `ostream` object.

In addition, note that the function returns a reference to an `ostream` object; this will be important in a second.

Now, for a few reasons which we won't go into right now, you do not have the ability to add additional such functions to the `ostream` class. Instead, all user-defined types are written to `ostream` objects via an overloaded `operator<<` that takes *two* parameters – the object you want to write to the stream, and the stream itself. For example, look at the `String` class we have been providing you:

```
friend ostream& operator<<(ostream& Out, const String& outputString);
```

This function takes as parameters the `String` object that you want to write to the `ostream` object, and it *also* takes the `ostream` object itself. In particular, it takes a reference to the `ostream` object, so that you are writing to the actual `ostream` object you passed in, and not to a copy of it as would have been the case if we used pass-by-value. In addition, since this function is associated with the `String` class, we need to make it a `friend` of that class in order to allow this function to have access to the private data of `String`. After all, we cannot properly print out the data unless we have access to it in the first place!

Note that this function also returns a reference to an `ostream` object; again, this will be important in just a bit.

Now, take a look at the implementation of this function in `string.C`:

```
ostream & operator<<(ostream& Out, const String& outputString)
{
    Out << outputString.stringArray;
    return Out;
}
```

Since `stringArray` is of type `char*`, writing this value to an output stream can be done by the built-in functions of the `ostream` class. So, we could have written the function as follows:

```
ostream & operator<<(ostream& Out, const String& outputString)
{
    Out.operator<<(outputString.stringArray);
    return Out;
}
```

and it would work equally well, since the first version is basically just the "easy syntax" which is allowed due to the fact that `<<` is an operator.

So, all we are doing in our implementation is writing the internal data of our `String` object to the `ostream` object called `Out`. And, since this is just a reference to the `ostream` object that we passed to this function in the function call, we are really writing the `String` member data to the `ostream` object that we passed to this function in the function call. Therefore, if we have code like this:

```
String s1("hi");
cout << s1;
```

then what we are really doing is making the following function call:

```
String s1("hi");
operator<<(cout, s1);
```

Then, in the `String` friend function `operator<<`, the variable `Out` is a reference to `cout` – i.e. a second name for the object named `cout`. The statement:

```
Out << outputString.stringArray;
```

is therefore in this case accomplishing the same purpose as the statement:

```
cout.operator<<(outputString.stringArray);
```

due to the fact that `Out` is a reference to `cout`. And, therefore, you print the member `char*` of your `String` object to the screen!

Finally, just as you can chain assignments or additions together:

```
a = b = c = d;          x = a + b + c + d;
```

you can do the same with stream output. This is where the fact that our `operator<<` functions return references to `ostream` objects becomes so important. Code such as the following, which prints out two integers and the `endl` (which is just a character and is thus handled by the `ostream` member functions):

```
int x = 1;
int y = 2;
cout << x << y << endl;
```

is really the following:

```
   int x = 1;
   int y = 2;
   ((cout.operator<<(x)).operator<<(y)).operator<<(endl);
//   ____ ^^^^^^^^^^
//      first call (--- is object, ^^^ is function)
//
//
//  -------------------- ^^^^^^^^^^^
//       second call
//
//
//
// ----------------------------------- ^^^^^^^^^^^^
//        third call
```

Since each call returns the `ostream` object by reference, once you manipulate `cout` with the first call, you then return it – and again manipulate the newly-altered cout with your *second* call, and then return it from your second call and manipulate it *again* with your *third call*. User-defined types work the same way, except it is the two-parameter `operator<<` you are calling and not the one-parameter `operator<<`. The code:

```
   String s1("one");
   String s2("two");
   String s3("three");
   cout << s1 << s2 << s3;
```

is really:

```
   String s1("one");
   String s2("two");
   String s3("three");
   operator<<(operator<<(operator<<(cout, s1), s2), s3);
//                        ^^^^^^^^^^ ____  --
//                  first call (---- = parameters, ^^^ = function)
//
//
//             ^^^^^^^^^ --------------------  --
//                         second call
//
//
// ^^^^^^^^^^^ -----------------------------------  --
//                  third call
//
```

You can mix built-in and user-defined types, as you have seen. The code:

```
   int x = 1;
   int y = 2;
   String s1("three);
   String s2("four);
   cout << s1 << x << s2 << y;
```

is really the code:

```
   int x = 1;
   int y = 2;
   String s1("three);
   String s2("four);
   operator<<(operator<<(cout, s1).operator<<(x), s2).operator<<(y);
//             ^^^^^^^^^^ ____  --
//         first call (---- = parameters, ^^^^^ = function)
//
//
//            ____________________ ^^^^^^^^^^
//         second call (----- = ostream object, ^^^^ = member function
//
//
// ^^^^^^^^^^ ________________________________  --
//          third call (---- = parameters, ^^^^^ = function)
//
//
// ____________________________________________ ^^^^^^^^^^^^
//           fourth call (----- = ostream object, ^^^^ = member function
//
```

But the first version is much cleaner and therefore much nicer! And all we need to do to continue using the first version is to overload `operator<<` for our classes, so that we have the ability to write an object of our user-defined type to an output stream.

The system can call the functions in the correct order above based on the fact that a function's arguments need to be evaluated before the function itself can be evaluated (which is why call 2 happens before call 3) and on the fact that you must have your object – even if it is an object returned by a function (as `operator<<` returns an `ostream`) – before you can call a function on that object (which is why call 1 happens before call 2 and why call 3 happens before call 4). You can't complete call 4 until you have the object the function is being called on, which means you must complete call 3 first, but you can't do that until the first parameter is evalutated, which means you must complete call 2 first, but you can't do that until you have the object that function is being called on, so you must complete call 1 first.

Input streams will work the same way, except we use `operator>>` instead of `operator<<`, and our `operator>>` functions write data *from* the stream, `to` a variable, instead of the other way around. For example, the code:

```
   int x;
   cin >> x;   // reads in an integer from the keyboard input
```

is really the code:

```
int x;
cin.operator>>(x);   // reads in an integer from the keyboard input
```

which uses the `istream` member function

```
istream& operator>>(int n);
```

Just as `operator<<` returns an `ostream` reference, `operator>>` returns an `istream` reference. The two-parameter version – for, say, the `String` class – would be:

```
friend istream& operator>>(istream& In, String& inputString);
```

5 Command-line arguments

Now, before we proceed with our discussion of streams, it is necessary to pause for a moment and quickly discuss how to use command-line arguments with a file. Up to this point, we have sent input to our programs by way of directing a data file into an executable file:

```
a.out < test.1
```

This works fine if we had only one file of data, but it is certainly possible to have more as well. What if we wanted to pass in 5 files of data at once? This is not going to work if we can only use the above technique. However, we *can* use what are known as *command-line arguments* as well. An example of this would be as follows:

```
a.out test.1 test.2 test.3 test.4 test.5
```

In the above line, we have 5 arguments to `a.out` – namely, the test files 1 through 5. In a sense, you can think of `a.out` as a function – because when it is first invoked, you start at `main()`, which *is* a function.

The idea is that the function `main()` can take any number of "parameters", not by actually having vast numbers of parameters, but by having exactly two parameters. A `main()` for a program that takes command-line arguments starts as follows:

```
int main(int argc, char* argv[])
{
       ...
```

The first parameter, `argc`, is the number of strings on the command-line that invoked this function. In this count, the string `a.out` counts, so in our example above, `argc` would have the value 6. The other parameter, `argv`, is an array of character pointers. And, as you may recall, in C++ a character pointer, i.e. a `char*` variable, can hold the starting address of a character array. So, this means that the array `argv` is an array of character array starting addresses, i.e. an array of C++ built-in strings. These strings will correspond exactly to the strings on our command-line. So, `argv[0]` has the value "`a.out`". `argv[1]` has the value "`test.1`". `argv[2]` has the value "`test.2`". And so on.

So, when we begin our program with the line:

```
a.out test.1 test.2 test.3 test.4 test.5
```

the first thing that happens is that the number of command line strings is counted, and stored in `argc`. Second, an array of `char*` cells indexed from 0 through `argc-1` is created, and given the name `argv`. Finally, an internal character array is created to hold each string, and the starting addresses of all those strings are stored in the corresponding cells in the `char*` array. Then, `argc` and `argv` are passed to `main()` and the program begins. Once the program ends, the system then eliminates these internal variables as well.

While in `main()`, you can refer to `argc` just as if it was an integer variable you declared yourself, and you can refer to `argv[0]` or `argv[4]` and obtain a `char*` which is the starting address of the array holding the appropriate command-line string. With this ability, you can convert the command-line string to whatever is appropriate – for example, you could pass in the `char*` to a `String` constructor, and you would then have a `String` object in your program whose value was the same as one of the command-line arguments you used when you started your program.

6 File streams

Now, although you are given `cout` and `cin` by the system, you generally will not directly declare your own `ostream` and `istream` objects. However, you *will* often create more *specific* versions of those objects, by taking a class derived from the class `istream` or the class `ostream` and declaring an object of *that* class. And, in fact, there are some classes in the system that are classses derived from `istream` or `ostream`, and so if you declare object of such types, you have objects which are more specific versions of `istream` or `ostream` objects.

Two such classes provided by the system are the classes `ifstream`, which is derived from `istream`, and `ofstream`, which is derived from `ostream`. You can use these classes by including the file `fstream.h` in your program. The class `ifstream` is a file input stream class, and objects of this type are streams that allow you to read values out of a file and into memory. Likewise, the class `ofstream` is a file output stream class, and objects of this type are streams that allow you to write values from memory into a file.

We will first examine the `ifstream` class. Take a look at the `main.C` for this MP. At one point, you see the following code:

```
ifstream textfile(argv[1]);

if (!textfile)              // textfile didn't open properly
   cout << "file error!" << endl;
else                        // textfile DID open properly
{
   while (textfile >> newString)  // as long as the read-in works
   {
      s1 = String(newString);   // cast char* to String
      w1.InsertWord(s1);
   }
}
delete newString;   // clean up the memory we used
textfile.close();   // when we are done with filestream, close it
```

The first line is a declaration of an `ifstream` object. Just like any other declaration, you first have the type (`ifstream`), and then the name of the variable (`textfile`), and then any paramters that you wish to pass to the constructor (`argv[1]`). The `ifstream` constructor takes a `char*`

variable, which is what we passed it above, so the value we are passing the constructor does indeed match the `ifstream` constructor parameter type.

What should be stored in this `char*`? Well, what that constructor is trying to do is initialize a file input stream object. That is, a file input stream object is created by the declaration, and the end of that stream is attached to memory (i.e. to your program), because it is an input file stream. But, the start of this stream is supposed to be attached to a file, so that you can read from a file to memory. Which file should we attach it to? Well, the name of that file is what we are passing into the `ifstream` constructor!

If the file exists and can be read from, then the `ifstream` object is initialized successfully, and when the constructor call has finished, `textfile` is a file input stream variable that we can pass to `operator>>` functions. But, wait, how can that be? The function `operator>>` didn't have an `ifstream` parameter, it had an `istream` parameter. Well, yes, but remember that the class `ifstream` is derived from the class `istream`. So, if an object that is of type `ifstream` is passed into a function that takes a parameter of type `istream`, the system can *cast* the `ifstream` object upward and view it as an `istream` object instead. So, there is no need to rewrite all our `operator>>` functions to accept `ifstream` objects. We already have `operator>>` functions that accept `istream` objects, and if pass one an `ifstream` object (or an object of any other class derived from `istream`), the system can cast it to an `istream` object and the `operator>>` function can be used properly.

What if the file does not exist, though? Or, what if the file cannot be read from, or there are other such restrictions? Well, in those cases, the `ifstream` object (called `textfile` here) will internally record that there was a failure trying to initialize the stream, and that the stream is now in the "fail" state. In this case, nothing can be read from the file, because input operations do nothing if the input stream is in the "fail" state.

So, how can we tell whether the `ifstream` object was opened successfully or not? Well, there is a member function called `operator!()` which is a unary operator – that is, it works on one value, not two like, say, addition would. This function will return 1 if the `ifstream` object is in the "fail" state, and 0 if it is not in the "fail" state. Just as you are used to using "!" to mean "not" or to reverse a boolean value (i.e. `if (!(s1==s2))` means "if it is NOT true that `s1` and `s2` are equal"), likewise the same usage is in effect here. When we use the line `if (!textfile)` above, we are saying, "if it is NOT true that `textfile` can be used correctly...". More specifically, we are asking, "if `textfile` is in the "fail" state...". Then, if the `if` case is true, that means that `operator!()` returned true, which means that the `ifstream` object was indeed NOT initialized correctly, and so our action in this case is to run some error handling code. Otherwise, the `ifstream` object *was* initialized correctly, which means that `textfile` is now a file input stream that runs between our file and memory, and so in this case we can proceed to read from the file into memory (our `else` case).

Next, we come to the line `while (textfile >> newString)`. Here, we are using the `operator>>` function, and since `newString` is a `char*` variable (see `main.C`), we are reading a single text string from `textfile` into the character array `newString`. `operator>>` then returns an `istream` object, just as `operator<<` returns an `ostream` object. What happens if the read is unsuccessful (for example, if we were at the end of the file and tried to read another string)? Well, the `while` condition will be 0, and so the `while` loop will exit and will not attempt to read or process another string.

But, wait, that doesn't quite work. When we look at the `while` loop, we understand it to mean:

```
while (textfile.operator>>(newString))
```

But, we said that `operator>>` returned an `istream` object. An `istream` object does not have values such as NULL and non-NULL, or 0 and 1, and therefore it can't be "tested" as the condition of a `while` loop. So how on earth does the above statement make any sense?

The answer is that, even though it looks like there is only *one* function call on this line, there are actually `two` function calls on this line. First, the `operator>>` is being called, and second, the `while` condition is being checked for truth or falsehood. Now, since there is no `!` as there was when we initialized the `ifstream` object, it is a little harder to tell there are two operations here. There is no visual cue to make it clear – you simply need to use your knowledge of programming to know that there has to be a check of a condition here (since it is a `while` loop) but that input is also going on (since you can clearly see the call to `operator>>` due to the presence of the stream object and the `>>` symbol).

So, we can't see two separate function calls, but we know there are two separate function calls there. How can that be? Well, the second operation is a *cast* or *conversion*. Such conversions happen all the time. We spoke of one above – `ifstream` objects can be converted to `istream` objects when needed. As another example, if you have a variable of type `int`, and you assign it a `float` value, the `float` will be converted to an `int` (via truncation) before being stored in the `int` variable:

```
int x;
x = 3.6;    // float 3.6 is converted to
            //      integer 3 before being stored in x
cout << x;  // will print out 3
```

We could have made this explicit via a *cast*

```
int x;
x = (int) 3.6;    // float 3.6 is converted to
            //      integer 3 before being stored in x
cout << x;  // will print out 3
```

but the language can handle certain casts, such as the one above, automatically without our explicitly saying to make such a cast.

How can this be done? Well, the language recognizes that you have an integer variable, and that you have a `float` value you are trying to assign to it. So, it sees if it has any way of converting floats to integers, and it does. This seems pretty handy, and perhaps user-defined functions could benefit from it as well. For example, it might be handy to be able to convert a `String` value to an `int`, so that we could perform numerical calculations on the values of `Strings` that held only integers, i.e.:

```
String s1("2100");
String s2("3100");
int x = (int) s1  +  (int) s2;  // x now stores 5200
```

Can we do this? Yes! Well, a conditional yes. We need to have added a specific function to the `String` class that *allows* us to do this. And that function is known as a *conversion operator*:

```
String::operator int()
```

This is a member function of the `String` class (well, it *would* be if we had added it to the `String` class) that will read the string – the private data of the class – and convert that string into an integer that is then returned as the return type. So, the expression `(int) s1` above generates the result `2100` from the `String ``2100''`. The actual `String` object is not necessarily changed; it depends how we have coded the conversion operator. All the operator really does , given a

`String` object that it is called on, is return an `int`. *Which* integer is returned is up to us and our implementation. At any rate, a conversion operator takes no parameters, you don't need a return type (and note that we don't have one) because the return type is implicit in the function name (`int` in this case), and the compiler can usually just call these automatically in an attempt to get our types to match correctly.

Now this might be very confusing; if it is, don't worry. I am just trying to give everyone a very clear picture of what is going on here, but understanding this idea is not critical to understanding streams.

At any rate, that is how we can have the `istream` object in the `while` loop and still get an actual condition check. The answer is, that there is a conversion, to a `void*`. That is, the class `istream` has a member function called

```
operator void*()
```

which, given the `istream` object that this function is called on, returns a `void*`, i.e. a type-less pointer. What value does this pointer hold? Well, it holds NULL if the `istream` object is in the "fail" state, and it holds a non-NULL value otherwise. The compiler finds this function during its desperate search to attempt to find some way of converting the `istream` returned by `operator>>` to a type that holds a 0/1 or NULL/non-NULL value. So, when we see:

```
while (textfile >> newString)
```

it is really the code:

```
while ( (void*)  (textfile.operator>>(newString)))
```

and *there* is our second function call. First, `operator>>` is called to do the actual input, it returns the `istream` object (which as we know is really an `ifstream` object), and then `operator void*()` is called to generate a NULL or non-NULL value from this which is then used in the `while` condition check.

Whew! Like I said above, this is all somewhat difficult and you don't really need to understand how it all works to use streams. It is enough to know that

```
while (textfile >> newString)
```

will attempt to read another string from `textfile` into `newString`, and that the `while` condition will be true if this is successful, and false if it it not. That is *all* you need to know. But, if you were inclined to compare the types in your head and realize something doesn't make sense, well, that is what is going on, and so hopefully now it makes sense. What this means it that we can use stream objects as conditions in `while` loops and `if` statements and such, as follows:

```
// assume we have a stream object called myStream

if (myStream)              OR    while(myStream)
{                                {
   ...                                 ...
```

means the condition will be true and we will continue as long as the stream is in a legal rather than a "fail" state. This is accomplished using the `operator void*()` conversion operator. On the other hand, we can also do things the other way around:

```
// assume we have a stream object called myStream

if (!myStream)            OR     while (!myStream)
{                                {
    ...                                  ...
```

means the condition will be true and we will continue as long as the stream is in a "fail" rather than a legal state. It could be in the "fail" state due to an unsuccessful initialization, or due to being out of elements to read, or other things could force the stream into a "fail" state as well. This check is accomplished using the `operator!()` that we used earlier above, and you will notice that this is exactly what we did when using `textfile` in the `if` statement near the start of `main.C`.

So, anyway, now that we know what is going on in the `while` condition, we can see that we will continue to run through the `while` loop until there are no more strings to input. At that time, we will attempt to read a string, find that we are all out of strings, the `while` condition will become false, and we will exit the while loop.

The last function to worry about, then, is one which closes the file input stream, signifying that we are done reading from this file. That function is `close()`, which is a member function of `ifstream`. You can see the usage of this in the line `textfile.close();`, which appears after the `while` loop in `main.C`. Closing the stream is the only thing this function does; there is nothing else to worry about and thus nothing else to discuss about this function.

So, that is the detailed description of what is going on in the file input part of `main.C`. There is a great deal more complexity to file input streams, and input streams in general, but you don't need to worry about anything more than we've already given you, and so we won't give you any more information right now. Deitel Chapter 14 has more info on file streams if you are interested, and Deitel Chapter 11 talks more about input and output streams in general.

As far as file output streams go, they are used in a very similar fashion. You declare an `ofstream` object in the same way you declared an `ifstream` object, namely by using `ofstream` as the type, and passing in a `char*` variable to the constructor. In the case of file output streams, though, this file is the name of a file you are going to *write* to. If the file does not exist, then such a file is created. If the file does exist, and you send nothing more to the `ofstream` constructor than the file name, the existing file is for all conceptual purposes removed and re-created. That is, you need to pass in additional values to the `ofstream` constructor if you want to create a stream that can *append* to a pre-existing file. We are not going to worry about this right now, so you can assume that each time you create a file output stream, you are creating a brand-new file with nothing in it and streaming to that file. This means you should just pass a `char*` file name to the `ofstream` constructor, and nothing else.

Just as you read from a file input stream by using `operator>>`, you write to a file output stream by using `operator<<`. Keep in mind that `ofstream` is a class that is derived from the class `ostream`, and that means that you can cast `ofstream` objects upward to `ostream` objects when needed. For example (AND READ THIS CAREFULLY BECAUSE IT IS A VERY HELPFUL HINT!!!), the `String operator<<` can be used to write a `String` object to an `ofstream` object, because you will simply be making the call as follows:

```
// assume textfile is an ofstream object and s1 is
//    a String object

// your code line
textfile << s1;
```

```
// which really calls operator<< as follows:
//   operator<<(textfile, s1);
```

and the `ofstream` object called `textfile` will be treated as an `ostream` object by the `operator<<` function.

Finally, `ofstream` objects should be closed when you are done with them, using `close()`, just as with `ifstream` objects.

That is more or less everything you need to know about file streams in order to complete the MP. The file `main.C` gives you examples of all of this stuff in the context of file input streams, and you can use the same ideas when you read from a file using streams, and can use similar ideas when you write to a file using streams.

7 Your assignment

Your task is to write a class called WordCounter, which will be located in the files `wordcounter.h` and `wordcounter.C`.
Private data: a single data member, which will be a binary search tree object. This object will hold objects of type KeyPair, but only very specific KeyPairs – KeyPairs which hold Strings for keys and ints for information.
Once you have looked over the keypair.* files and the bstree.* files, the above paragraph should be sufficient to allow you to:

1. determine how to declare the member data BST,
2. determine how to fill in its template type, and
3. determine whether or not WordCounter is a template (though (3) is also blatantly obvious from main.C, but try to figure it out from the above description first).

Public functions:

- Constructor - not much to do, other than to handle the initialization of the member BSTree. Use the initializer list for this – that is, call the default constructor for the BSTree object on the initializer list of this function. For examples of initializing members on the initializer list, look at the TreeNode class constructors and the Tree constructor in bstree.*
- InsertWord - takes a String as a parameter. If this String has been inserted before, then the information integer associated with it should be increased by 1. Otherwise, the integer "1" should be associated with this String. The BST handles the storage of the association pairs.

  Note that this will not be anywhere near as straightforward as it looks, because the ADT of the BSTree and KeyPair classes limit what you can do. You will have to find a way to handle the updating of the information stored in a KeyPair in the BStree using the interface functions you are given.

  You can alter the tree however you like in order to do this. All we care about is that you implement the specification, not what the tree looks like. (Note: I do NOT mean you can alter the BSTree *class*. I just mean that you can alter the BSTree object stored as a member data of your class.)

  You will learn to appreciate a well-written interface after being restricted somewhat in your attempts to code this function. :-)

- PrintFullInfo - prints all Strings stored in the tree, in alphabetical order, along with their associated integers. See test.x and test.x.std files for examples of how the test data gets printed out in the end.
- PrintWordInfo - takes a String parameter. If String is in tree, print out String and its associated info. If it is not, print out String and the number 0. See the top of test.x.std for examples.
- ReadFromFile - takes a char* which will serve as a file name. create an ifstream similar to the way we did this in main.C, and read in the information from the file.
- WriteToFile - takes a char* which will serve as a file name, create an ofstream object. This stands for "output stream" just as in main.C we used an "input stream". We can write to such an output stream the same way we write to cout. Write out the data in this object.
- *** note that in main.C, you read from the file you wrote. So, you can write whatever you want to the file, as long as you can parse it correctly in the read function.
- *** note also that when you write to the file, you will not necessarily be able to reproduce the exact same BSTree structure when you read from the file into an object. You want the same *data*, but the internal arrangement of the BSTree TreeNodes is not something you care about.

8 PSP

Mattox is just about done with his PSP program and will be posting instructions concerning how to use it.

9 Handing in your code

To handin your MP5 code, use the command

```
handin cs225 mp5 wordcounter.h wordcounter.C
```