CIS 291 – Data Structures in C++ - Spring 2005
Homework #8
HW #8 due: FRIDAY, APRIL 1st, 5:00 pm

(that's not an April Fool ---
it is due 1 day later due to the Cesar Chavez holiday on Thursday, March 31st)

Purpose: more linked structure practice, using linked-list mergesort, and using a list ADT with a non-builtin class.

How this will be turned in:
Use **~st10/291submit**, called from the directory on cs-server where the files you wish to submit are stored.

Homework #8

1. We discussed a general **list** ADT informally in lecture. Consider the simpler **babylist** ADT provided along with this homework handout, and implement it as a class. (For this homework, it is not required to be a template class, but it may be if you wish it to be.)

Additional requirements:

* you must implement your **babylist** ADT as a singly-linked list, using the **node** class posted along with HW #7.

* in implementing the **sort** operation within the **babylist** ADT pseudo-UML, you must use call the **mergesort** function discussed in class, and also available from the course web page. (Careful --- which files do you need to copy over? Which files need to be #include'd in **babylist.cpp/babylist.template**? Which files need to used in the **g++** command creating an executable for a main function using the **babylist** class? (hint: it's a bunch... more support for makefiles!))

After you have written **babylist.h** and **babylist.cpp** (or **babylist.template**, if you chose to implement a template class), write a testing main function **test_babylist.cpp** to list your class, and redirect its output into **291hw08_1_out**. (Although your program also uses the node and mergesort files, these should not need to be modified, and so they do not need to be submitted.)

You will submit your versions of **babylist.h**, **babylist.cpp**, **test_babylist.cpp**, and **291hw08_1_out**.

2. Now, consider the **wordcount** class from HW #6. You could have a **babylist** of wordcount instances; you could sort a linked list of wordcount instances --- or, you could with a few more features added to the wordcount class.

First of all, **mergesort** expects to be able to compare the data fields; it happens to use **<=** . So, **wordcount** needs to include this operator. The decision has been made the **wordcount**'s **<=** makes the decision based on the involved instance's **count** field values.

Since I cannot remember if **CIS 230** includes coverage of implementing your own operators, this will be included this time (either as reminder or as an example, whichever the case may be):

Within the **wordcount.h** class declaration, you should add an **OPERATORS** section comment and the code:

```
// postcondition: returns true if this object's count field is less than
//    or equal to <operand>'s count field
//
bool operator <= (wordcount operand);
```

Within the **wordcount.cpp** implementations, you should add an **OPERATORS** section comment and the code:

```
// postcondition: returns true if this object's count field is less than
//    or equal to <operand>'s count field
//
bool wordcount::operator <= (wordcount operand)
{
    return ( get_count() <= operand.get_count() );
}
```

The above will permit linked lists of nodes whose data field contains **wordcount** instances to be sorted in order of ascending count field values.

We need another more thing --- there are times when C++ automatically calls a default constructor for a class. I'd avoided specifying one for **wordcount**, because the intent is that the string field cannot be changed once it is initialized. But, it seems to need to be there, even though you cannot really do much with such an instance. So, a default operator needs to be added under the CONSTRUCTORS section in wordcount.h:

```
// postcondition: creates a wordcount instance with word that
//    is an empty string and a count of 0
// (here as a concession to when C++ requires the presence of a
//    default constructor --- note that word fields are not intended
//    to be changeable once initialized)
//
wordcount();
```

And, in the CONSTRUCTORS section of wordcount.cpp (but using **your** field names...!):

```
// postcondition: creates a wordcount instance with word that
//    is an empty string and a count of 0
// (here as a concession to when C++ requires the presence of a
//    default constructor --- note that word fields are not intended
//    to be changeable once initialized)
//
wordcount::wordcount()
{
    word_field = "";
    count_field = 0;
}
```

Finally --- sadly --- I wrote #3 assuming you'd be able to print the **value_type** used in **babylist**. That means wordcount needs to be able to be printed using **cout**!!

To get this capability, you must add a **nonmember function** implementing operator **<<** for wordcount. In **wordcount.h** , **AFTER** the class definition but BEFORE the #endif, put a section comment for NONMEMBER FUNCTIONS and then:

```
// postcondition: outputs the data fields of a wordcount instance
//    <source> relatively gracefully to ostream <outs>.
//
ostream& operator <<(ostream& outs, const wordcount& source);
```

And, **wordcount.cpp**, first #include **<iostream**>, and then add a section comment at the end for NONMEMBER FUNCTIONS and then:

```
// postcondition: outputs the data fields of a wordcount instance
//    <source> relatively gracefully to ostream <outs>.
//
ostream& operator <<(ostream& outs, const wordcount& source)
{
    outs << "word: " << source.get_word()
         << " count: " << source.get_count();
    return outs;
}
```

Modify your **test_wordcount.cpp** function to appropriately test the new constructor, new operator, and <<, and redirect the results of your testing into **291hw08_2_out**.

Submit your modified **wordcount.h**, **wordcount.cpp**, **test_wordcount.cpp**, and **291hw08_2_out**.

3. Now, let's use both of the pieces from HW #1 and HW #2 in a simple application, **orderThem**.

I know that you covered file i/o in CIS 230. You are going to make use of this here.

Assume that a file contains input structured **exactly** as follows (it isn't your application's job to verify this, or complain if it isn't so --- it may simply blithely assume that it is): on the first line is an integer, representing how many words are in the file, on the next line is the first word, on the line after that is an integer representing its frequency, on the line after that is the next word, on the line after that is an integer representing the 2nd word's frequency, and so on, for as many words are were indicated on the 1st line.

What shall your application do? It will:

* ask the user for the name of an input file,
* try to open it, and if it can,
  then read the file,
  creating a wordcount instance for each word and count encountered
  and adding it to a **babylist** of wordcount instances.

* Then, sort the resulting babylist,
* and print out the babylist contents to the screen.

Now, it will be tricky to redirect **orderThem**'s output, because it has interactive input. The trick is that the prompt for entering the file name will be redirected to the output file, and you have to remember to simply type the input file name without actually being prompted for it (your program will patiently wait until you do...!) BUT, you can get the required output from an example run into a file **291hw08_3_out** in this way.

(This application is not a testing main --- so we won't be able to include what should be seen, and what is seen. Oh well...)

You will submit **orderThem.cpp**, your test input file, and **291hw08_3_out**.

When you are done, submit the following files using **~st10/291submit**:

babylist.h, **babylist.cpp**, **test_babylist.cpp**, **291hw08_1_out**.
wordcount.h, **wordcount.cpp**, **test_wordcount.cpp**, **291hw08_2_out**.
orderThem.cpp, your test input file, and **291hw08_3_out**.