`$Id: asg4-listmap-templates.mm,v 1.12 2010-02-26 14:21:11-08 - - $`
`/afs/cats.ucsc.edu/courses/cmps109-wm/Assignments/asg4-listmap-templates`

1. Overview

In this assignment, you will implement template code and not use any template classes from the STL. You will also write your own code to handle files. Refer to the earlier assignment as to how to open and read files. You may use **`<cassert>`**, **`<cerrno>`**, **`<cstdlib>`**, **`<exception>`**, **`<fstream>`**, **`<iomanip>`**, **`<iostream>`**, **`<stdexcept>`**, **`<string>`**, **`<typeinfo>`**, and if you think anything else is needed, post a question to the newsgroup. Specifically, you may not use any classes that take template parameters, such as **`<iterator>`**, **`<list>`**, **`<map>`**, **`<pair>`**, **`<vector>`**, except for those you write yourself.

2. Program Specification

The program is specified in the foramt of a Unix **`man`**(1) page.

NAME

keyvalue — manage a list of key and value pairs

SYNOPSIS

`keyvalue` [**`-@`** *flags*] [*filename . . .*]

DESCRIPTION

Input is read from each file in turn. Before any processing, each input line is echoed to **`cout`**, preceded by its filename and line number within the file. The name of **`cin`** is printed as a minus sign (**`-`**).

Each non-comment line causes some action on the part of the program, as described below. Before processing a command, leading and trailing white space is trimmed off of the key and off of the value. White space interior to the key or value is not trimmed. When a key and value pair is printed, the equivalent of the format string used is **`"%s = %s\n"`**. Of course, use **`<iostream>`**, not **`<stdio>`**. The newline character is removed from any input line before processing. If there is more than one equal sign on the line, the first separates the key from the value, and the rest are part of the value. Input lines are one of the following :

`#`

Any input line whose first non-whitespace character is a hash (**`#`**) is ignored as a comment. This means that no key can begin with a hash. An empty line or a line consisting of nothing but white space is ignored.

key

A line consisting of at least one non-whitespace character and no equal sign causes the key and value pair to be printed. If not found, the message

key: **`key not found`**

is printed. Note that the characters in italics are not printed exactly. The actual key is printed. This message is printed to **`cout`**.

key **`=`**

If there is only whitespace after the equal sign, the key and value pair is deleted from the map.

key **`=`** *value*

If the key is found in the map, its value field is replaced by the new value. If not found, the key and value are inserted in increasing lexicographic order, sorted by key. The new key and value pair is printed.

`=`

All of the key and value pairs in the map are printed in lexicographic order.

`/=`

All of the key and value pairs in the map are printed in reverse lexicographic order, i.e., from last to first.

`=` *value*
All of the key and value pairs with the given value are printed in lexicographic order sorted by key.

OPTIONS

The `-@` option is followed by a sequence of flags to enable debugging output, which is written to the standard error. The option flags are only meaningful to the programmer.

OPERANDS

Each operand is the name of a file to be read. If no filenames are specified, `cin` is read. If filenames are specified, a filename consisting of a single minus sign (`-`) causes `cin` to be read in sequence at that position. Any file that can not be accessed causes a message in proper format to be printed to `cerr`.

EXIT STATUS

0 No errors were found.

1 There were some problems accessing files, and error messages were reported to `cerr`.

3. Implementation Sequence

In this assignment, you will constuct a program from scratch, using some of the code from previous assignments.

(a) Study the behavior of `code/pkeyvalue.perl`, whose behavior your program should emulate. The Perl version does not support the debug option of your program.

(b) Copy `Makefile` from your previous assignment, and edit it so that it will build and submit your new assignment.

(c) Copy the files `trace.{h,cc}` and `util.{h,cc}` from one of your earlier assignments, adding code to them as necessary.

(d) Implement your main program whose name is `main.cc`, and handle files in the same way as the sample Perl code. Instead of trying to use a map, just print debug statements showing which of the five kinds of statements are recognized, printing out the key and value portion of the statement.

(e) Instead of `<pair>` from the STL, you will use `pairx.{h,cc}`.

(f) You will be using a linear linked list to implement your data structure. This is obviously unacceptable in terms of a real data structures problem, since unit operations will run in $O(n)$ time instead of the proper $O(\log_2 n)$ time for a balanced binary search tree. But iteration over a binary search tree is rather complex and will not contribute to your learning about how to implement templates. And balancing a BST is part of CMPS-101, which is not a prerequisite for this course.

(g) Look at `comparex.{h,cc}` and `testcmpx.cc`, which show how to create and use a `comparex` object to make comparisons. The `listmap` class assumes this has already been declared.

(h) The files `*.ccti` are explicit template instantiations that should not be necessary, since the compiler ought to be able to figure out these requirements themselves and automatically perform the instantiations. `/opt/SUNWspro/bin/CC` is capable of doing this, but `/usr/sfw/bin/g++` is not. These template instantiation files must be include from the `.cc` file ***after*** the definition of the template function. They are placed in separate files in order to keep the implementation file separate from excessive coupling between implementation and client modules.

4. Template class `listmap`

We now examine the class `listmap`, which is partially implemented for you. You need not implement functions that are never called.

`typedef pairx <key_t, value_t> mappairx;`
: is a pair to be kept in each node and returned and examined by an iterator.

`listmap ();`

`~listmap ();`
: The destructor must link down the list and `delete` each node in the list. It is assumed that the pair has its own destructors.

`class iterator {`
: is used to iterate down the list for those commands which require multiple line output. It is also used to find out if an element is in the list, and to erase it.

`friend class listmap;`
: A `listmap` may create an iterator, but not anyone else.

`iterator ();`
: The client class must not be permitted to construct a default iterator. This should ***not*** be implemented. The other three implicit members may be used.

`mappairx &operator* ();`
: dereferences an iterator into a reference pair. Only valid if the iterator points at an element in the list. Undefined if it is the value returned by `end()`.

`mappairx *operator-> ();`
: allows an iterator to be used to select the `first` and `second` fields of a `pairx`.

`iterator &operator++ ();`
: Increments the pointer, stepping one element down the list. Undefined if at the end of the list.

`iterator &operator-- ();`
: backs up the list by one element. Undefined for the value returned by `begin()`.

`bool operator== (const iterator &);`
: compares two iterators for equality.

`bool operator!= (const iterator &);`
: or inequality.

`void erase ();`
: causes the element indicated by the iterator to be erased from the list.

`struct node;`
: is used to represent the list.

`iterator ();`

`iterator (const iterator &);`

`iterator &operator= (const iterator &);`

`explicit iterator (node *where);`
: creates an iterator pointing at a particular node. Note that it is `explicit`.

`node *where;`
: is used by the iterator to remember which node it points at.

`listmap *map;`
: is used by the interator to find the map itself in the case that the head node is deleted.

`void insert (key_t, value_t);`
: Searches the list for a key and puts a new key and value pair into the list. If the key already exists, replaces the value instead of creating a new node. The list is kept in lexicographical order based on the `less` object.

`iterator find (key_t);`
Searches the list for a key and returns an iterator to it.

`iterator begin ();`
returns a pointer to the head of the list.

`iterator end ();`
Returns a value indicating the phantom element after the end of the list. Using `operator--` on this element gets at the last element of the list, if any.

5. What to Submit

`Makefile`, `README`, and all necessary C++ header and implementation files. Use the script `testsubmit` to verify that your program will compile. And don't forget `checksource`. If you are using pair programming, read and follow the instructions in `/afs/cats.ucsc.edu/courses/cmps109-wm/Syllabus/pair-programming` and submit `SCORE.pair`.