$Id: asg3-listmap-templates.mm,v 1.119 2022-02-17 10:10:21-08 - - $
/afs/cats.ucsc.edu/courses/cse111-wm/Assignments/asg3-listmap-templates
https://www2.ucsc.edu/courses/cse111-wm/:/Assignments/asg3-listmap-templates/

1. Overview

In this assignment, you will implement template code and not use any template classes from the standard library. 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 the following includes, and if you think anything else is needed, post a question to the discussion forum: **<cassert>**, **<cerrno>**, **<cstdlib>**, **<cstring>**, **<exception>**, **<fstream>**, **<iomanip>**, **<iostream>**, **<regex>**, **<stdexcept>**, **<string>**, **<typeinfo>**.

Specifically, you may not use any classes that take template parameters, such as **<iterator>**, **<list>**, **<map>**, **<pair>**, **<vector>**, except for those you write yourself. Do not use **shared_ptr**, and instead, explicitly manage pointers yourself using **new** and **delete**. Ensure that there are no dangling pointers and no memory leak. Use **valgrind** to check on this.

2. Program specification

The program is specified in the format 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 (**-**).

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**.

INPUT DESCRIPTION

Each non-comment line causes some action on the part of the program. 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"**. 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.

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

EXIT STATUS

0 No errors were found.

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

3. Reference implementation.

Study the behavior of **misc/pkeyvalue.perl**, whose behavior your program should emulate. The Perl version does not support the debug option of your program.

4. Implementation sequence

Consider this implementation sequence, but you may of course choose to implement your program in a different order.

(a) Implement your main program whose name is **main.cpp**, 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.

(b) Instead of **<pair>** from the standard library, you will use **xpair**.

(c) 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.

(d) Look at **xless.h** and **misc/testxless.cpp**, which show how to create and use an **xless** object to make comparisons. The **listmap** class assumes this has already been declared.

(e) The files ***.tcc** are explicit template instantiations. Templates are type-safe macros and the source is needed at the point where they are compiled.

(f) For the two queries requiring iteration, your main module must make use of the **listmap** iterators. Do not copy anything from the listmap into another data structure, such as a vector. That is, there may be no function of **listmap** which copies out all of the items into another data structure.

5. The main function

Replace the code in the main function to do options analysis. Then, for each input line, use **regex_search** using regular expressions to parse the line into one of the three kinds of lines described above. Use the field captures to extract the key and value fields. See the example **matchlines.cpp**, which shows how to use regular expressions from the **<regex>** library.

6. Template class `listmap`

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

(a)
```
template <typename key_t, typename mapped_t,
          class less_t=xless<key_t>>
class listmap
```
defines the template class with three arguments. **key_t** and **mapped_t** are the elements to be kept in the list. **less_t** is the class used to determine the ordering of the list and defaults to **xless<key_t>**.

(b)
```
using key_type = key_t;
using mapped_type = mapped_t;
using value_type = xpair<const key_type, mapped_type>;
```
are some standard names given to usual standard library types. Note that the value type is an **xpair**, not what is normally thought as the value, which here is called the mapped type.

(c) **struct link**
represents the list itself and is contained in every node. The list is kept as a circular doubly linked list with the list itself being the start and end, as well as the **end()** result. In a list with n nodes, there are $n+1$ links, each node having a link, and the list itself having a link, but not node values.

(d) **`struct node`**
is a private node used to hold a value type along with forward and backword links to form a doubly linked list. It inherits from **`struct link`**. The private function **`anchor()`** downcasts from a **`link`** to a **`node`**.

(e)
```
listmap();
listmap (const listmap&);
listmap& operator= (const listmap&);
listmap (listmap&&);
listmap& operator= (listmap&&);
~listmap();
```
The usual six members are overriden and explicitly defined.

(f) **`iterator insert (const value_type&);`**
Note that insertion takes a pair as a single argument. If the key is already there, the value is replaced, otherwise there is a new entry inserted into the list. An iterator pointing at the inserted item is returned.

Algorithm :

1. Chase down the list from **`begin`** until you find either **`end`** or a node with a key not less than the key you are searching for. Always use **`xless`** for comparisons.
2. If you hit **`end`**, insert in front of end.
3. Use **`xless`** with the operands reversed.
   - If they are equal, replace the value.
   - If not, insert the new node in front of the node just found.

You may only use **`xless`** to compare keys. Do not use any relational operator. If **`xless(a,b)`** and **`xless(b,a)`** are both false, that tells you they are equal.

(g) **`iterator find (const key_type&);`**
Searches and returns an iterator. If find fails, it returns the **`end()`** iterator.

As with **`insert`**, use only **`xless`** for comparisons. Do not use **`==`**, **`<`**, or other comparison operators on strings.

(h) **`iterator erase (iterator position);`**
The item pointed at by the argument iterator is deleted from the list. The returned iterator points at the position immediately following that which was erased.

An attempt to erase using an invalid iterator (including such as **`end`**) causes *undefined behavior*. Undefined behavior may corrupt memory, cause a segmentation fault, or cause other misery.

(i)
```
iterator begin();
iterator end();
```
The usual iterator generators. We don't bother here with a constant iterator. Since the list is circular, **`end()`** is just the list itself.

7. Template class `listmap::iterator`

Although the iterator is nested inside the list map, it is easier to read when specified separately. Using **`erase`**, **`operator->`**, or **`operator*`** on an invalid iterator (including **`end`**) causes *undefined behavior*. Undefined means that anything can happen. It

is the programmer's responsibility not to cause undefined behavior.

(a) `class listmap<key_t,mapped_t,less_t>::iterator`
specifies precisely which class the iterator belongs to.

(b) `friend class listmap<key_t,mapped_t>;`
Only a listmap is permitted to construct a valid iterator.

(c) `iterator (node* where);`
The iterator keeps track of the node.

(d) `value_type& operator*();`
Returns a reference to some value type (key and value pair) in the list. Selections are then by dot (`.`).

(e) `value_type* operator->();`
Returns a pointer to some value type, from which fields can be selected with an arrow (`->`).

(f) `iterator& operator++(); //++itor`
`iterator& operator--(); //--itor`
Move backwards and forwards along the list. Moving off the end with `++` and moving from an end iterator to the last element does not require special coding, since the list is a circular list. We don't bothere here with the postfix operators.

(g) `void erase();`
Removes the key and value pair from the list.

8. What to Submit

`Makefile`, `README`, and all necessary C++ header and implementation (`*.h`, `*.tcc`, `*.cpp`) files. If you are using pair programming, also submit `PARTNER`.

9. Diagram of listmap with data