

```
1: // $Id: listmap.h,v 1.11 2015-04-28 19:22:02-07 - - $
2:
3: #ifndef __LISTMAP_H__
4: #define __LISTMAP_H__
5:
6: #include "xless.h"
7: #include "xpair.h"
8:
9: template <typename Key, typename Value, class Less=xless<Key>>
10: class listmap {
11:     public:
12:         using key_type = Key;
13:         using mapped_type = Value;
14:         using value_type = xpair<const key_type, mapped_type>;
15:     private:
16:         Less less;
17:         struct node;
18:         struct link {
19:             node* next{};
20:             node* prev{};
21:             link (node* next, node* prev): next(next), prev(prev){}
22:         };
23:         struct node: link {
24:             value_type value{};
25:             node (node* next, node* prev, const value_type&);
26:         };
27:         node* anchor() { return static_cast<node*> (&anchor_); }
28:         link anchor_ {anchor(), anchor()};
29:     public:
30:         class iterator;
31:         listmap(){};
32:         listmap (const listmap&);
33:         listmap& operator= (const listmap&);
34:         ~listmap();
35:         iterator insert (const value_type&);
36:         iterator find (const key_type&) const;
37:         iterator erase (iterator position);
38:         iterator begin() { return anchor()->next; }
39:         iterator end() { return anchor(); }
40:         bool empty() const { return begin() == end(); }
41: };
42:
```

```
43:
44: template <typename Key, typename Value, class Less=xless<Key>>
45: class listmap<Key,Value,Less>::iterator {
46:     private:
47:         friend class listmap<Key,Value>;
48:         listmap<Key,Value,Less>::node* where {nullptr};
49:         iterator (node* where): where(where) {};
50:     public:
51:         iterator() {}
52:         value_type& operator*();
53:         value_type* operator->();
54:         iterator& operator++(); //++itor
55:         iterator& operator--(); //--itor
56:         void erase();
57:         bool operator== (const iterator&) const;
58:         bool operator!= (const iterator&) const;
59: };
60:
61: #include "listmap.tcc"
62: #endif
63:
```

```
1: // $Id: trace.h,v 1.2 2014-04-24 18:02:55-07 - - $
2:
3: #ifndef __TRACE_H__
4: #define __TRACE_H__
5:
6: #include <iostream>
7: #include <string>
8: #include <vector>
9:
10: using namespace std;
11:
12: //
13: // traceflags -
14: //     static class for maintaining global trace flags, each indicated
15: //     by a single character.
16: // setflags -
17: //     Takes a string argument, and sets a flag for each char in the
18: //     string. As a special case, '@', sets all flags.
19: // getflag -
20: //     Used by the TRACE macro to check to see if a flag has been set.
21: //     Not to be called by user code.
22: //
23:
24: class traceflags {
25:     private:
26:         static vector<char> flags;
27:     public:
28:         static void setflags (const string& optflags);
29:         static bool getflag (char flag);
30: };
31:
32: //
33: // TRACE -
34: //     Macro which expands into trace code. First argument is a
35: //     trace flag char, second argument is output code that can
36: //     be sandwiched between <<. Beware of operator precedence.
37: //     Example:
38: //         TRACE ('u', "foo = " << foo);
39: //     will print two words and a newline if flag 'u' is on.
40: //     Traces are preceded by filename, line number, and function.
41: //
42:
43: #define TRACE(FLAG, CODE) { \
44:     if (traceflags::getflag (FLAG)) { \
45:         cerr << "[" << __FILE__ << ":" << __LINE__ << ":" \
46:             << __func__ << "]" " << boolalpha; \
47:         cerr << CODE << endl; \
48:     } \
49: }
50:
51: #endif
52:
```

```
1: // $Id: util.h,v 1.4 2015-02-11 13:03:38-08 - - $
2:
3: //
4: // util -
5: //     A utility class to provide various services not conveniently
6: //     associated with other modules.
7: //
8:
9: #ifndef __UTIL_H__
10: #define __UTIL_H__
11:
12: #include <iostream>
13: #include <list>
14: #include <stdexcept>
15: #include <string>
16:
17: using namespace std;
18:
19: #include "trace.h"
20:
21: //
22: // sys_info -
23: //     Keep track of execname and exit status.  Must be initialized
24: //     as the first thing done inside main.  Main should call:
25: //         sys_info::set_execname (argv[0]);
26: //     before anything else.
27: //
28:
29: class sys_info {
30: public:
31:     static const string& get_execname ();
32:     static void set_exit_status (int status);
33:     static int get_exit_status ();
34: private:
35:     friend int main (int argc, char** argv);
36:     static void set_execname (const string& argv0);
37:     static string execname;
38:     static int exit_status;
39: };
40:
41: //
42: // datestring -
43: //     Return the current date, as printed by date(1).
44: //
45:
46: const string datestring ();
47:
48: //
49: // split -
50: //     Split a string into a list<string>..  Any sequence
51: //     of chars in the delimiter string is used as a separator.  To
52: //     Split a pathname, use "/".  To split a shell command, use " ".
53: //
54:
55: list<string> split (const string& line, const string& delimiter);
56:
```

```
57:
58: //
59: // complain -
60: //     Used for starting error messages.  Sets the exit status to
61: //     EXIT_FAILURE, writes the program name to cerr, and then
62: //     returns the cerr ostream.  Example:
63: //         complain() << filename << ": some problem" << endl;
64: //
65:
66: ostream& complain();
67:
68: //
69: // syscall_error -
70: //     Complain about a failed system call.  Argument is the name
71: //     of the object causing trouble.  The extern errno must contain
72: //     the reason for the problem.
73: //
74:
75: void syscall_error (const string&);
76:
77: //
78: // operator<< (list) -
79: //     An overloaded template operator which allows lists to be
80: //     printed out as a single operator, each element separated from
81: //     the next with spaces.  The item_t must have an output operator
82: //     defined for it.
83: //
84:
85: template <typename item_t>
86: ostream& operator<< (ostream& out, const list<item_t>& vec);
87:
88: //
89: // string to_string (thing) -
90: //     Convert anything into a string if it has an ostream<< operator.
91: //
92:
93: template <typename item_t>
94: string to_string (const item_t&);
95:
96: //
97: // thing from_string (const string&) -
98: //     Scan a string for something if it has an istream>> operator.
99: //
100:
101: template <typename item_t>
102: item_t from_string (const string&);
103:
104: //
105: // Put the RCS Id string in the object file.
106: //
107:
108: #include "util.tcc"
109: #endif
110:
```

```
1: // $Id: xless.h,v 1.3 2014-04-24 18:02:55-07 - - $
2:
3: #ifndef __XLESS_H__
4: #define __XLESS_H__
5:
6: //
7: // We assume that the type type_t has an operator< function.
8: //
9:
10: template <typename Type>
11: struct xless {
12:     bool operator() (const Type& left, const Type& right) const {
13:         return left < right;
14:     }
15: };
16:
17: #endif
18:
```

```
1: // $Id: xpair.h,v 1.4 2014-06-27 17:39:08-07 - - $
2:
3: #ifndef __XPAIR_H__
4: #define __XPAIR_H__
5:
6: #include <iostream>
7:
8: using namespace std;
9:
10: //
11: // Class xpair works like pair(c++).
12: //
13: // The implicitly generated members will work, because they just
14: // send messages to the first and second fields, respectively.
15: // Caution: xpair() does not initialize its fields unless
16: // First and Second do so with their default ctors.
17: //
18:
19: template <typename First, typename Second>
20: struct xpair {
21:     First first{};
22:     Second second{};
23:     xpair() {}
24:     xpair (const First& first, const Second& second):
25:         first(first), second(second) {}
26: };
27:
28: template <typename First, typename Second>
29: ostream& operator<< (ostream& out, const xpair<First,Second>& pair) {
30:     out << "{" << pair.first << ", " << pair.second << "}";
31:     return out;
32: }
33:
34: #endif
35:
```

```
1: // $Id: listmap.tcc,v 1.7 2015-04-28 19:22:02-07 - - $
2:
3: #include "listmap.h"
4: #include "trace.h"
5:
6: //
7: ///////////////////////////////////////////////////////////////////
8: // Operations on listmap::node.
9: ///////////////////////////////////////////////////////////////////
10: //
11:
12: //
13: // listmap::node::node (link*, link*, const value_type&)
14: //
15: template <typename Key, typename Value, class Less>
16: listmap<Key,Value,Less>::node::node (node* next, node* prev,
17:                                     const value_type& value):
18:     link (next, prev), value (value) {
19: }
20:
21: //
22: ///////////////////////////////////////////////////////////////////
23: // Operations on listmap.
24: ///////////////////////////////////////////////////////////////////
25: //
26:
27: //
28: // listmap::~~listmap()
29: //
30: template <typename Key, typename Value, class Less>
31: listmap<Key,Value,Less>::~~listmap() {
32:     TRACE ('l', (void*) this);
33: }
34:
```



```
35:
36: //
37: // iterator listmap::insert (const value_type&)
38: //
39: template <typename Key, typename Value, class Less>
40: typename listmap<Key, Value, Less>::iterator
41: listmap<Key, Value, Less>::insert (const value_type& pair) {
42:     TRACE ('l', &pair << "->" << pair);
43:     return iterator();
44: }
45:
46: //
47: // listmap::find(const key_type&)
48: //
49: template <typename Key, typename Value, class Less>
50: typename listmap<Key, Value, Less>::iterator
51: listmap<Key, Value, Less>::find (const key_type& that) const {
52:     TRACE ('l', that);
53:     return iterator();
54: }
55:
56: //
57: // iterator listmap::erase (iterator position)
58: //
59: template <typename Key, typename Value, class Less>
60: typename listmap<Key, Value, Less>::iterator
61: listmap<Key, Value, Less>::erase (iterator position) {
62:     TRACE ('l', &*position);
63:     return iterator();
64: }
65:
```

```
66:
67: //
68: ///////////////////////////////////////////////////////////////////
69: // Operations on listmap::iterator.
70: ///////////////////////////////////////////////////////////////////
71: //
72:
73: //
74: // listmap::value_type& listmap::iterator::operator*()
75: //
76: template <typename Key, typename Value, class Less>
77: typename listmap<Key, Value, Less>::value_type&
78: listmap<Key, Value, Less>::iterator::operator*() {
79:     TRACE ('l', where);
80:     return where->value;
81: }
82:
83: //
84: // listmap::value_type* listmap::iterator::operator->()
85: //
86: template <typename Key, typename Value, class Less>
87: typename listmap<Key, Value, Less>::value_type*
88: listmap<Key, Value, Less>::iterator::operator->() {
89:     TRACE ('l', where);
90:     return &(where->value);
91: }
92:
93: //
94: // listmap::iterator& listmap::iterator::operator++()
95: //
96: template <typename Key, typename Value, class Less>
97: typename listmap<Key, Value, Less>::iterator&
98: listmap<Key, Value, Less>::iterator::operator++() {
99:     TRACE ('l', where);
100:     where = where->next;
101:     return *this;
102: }
103:
104: //
105: // listmap::iterator& listmap::iterator::operator--()
106: //
107: template <typename Key, typename Value, class Less>
108: typename listmap<Key, Value, Less>::iterator&
109: listmap<Key, Value, Less>::iterator::operator--() {
110:     TRACE ('l', where);
111:     where = where->prev;
112:     return *this;
113: }
114:
```

```
115:
116: //
117: // bool listmap::iterator::operator== (const iterator&)
118: //
119: template <typename Key, typename Value, class Less>
120: inline bool listmap<Key,Value,Less>::iterator::operator==
121:         (const iterator& that) const {
122:     return this->where == that.where;
123: }
124:
125: //
126: // bool listmap::iterator::operator!= (const iterator&)
127: //
128: template <typename Key, typename Value, class Less>
129: inline bool listmap<Key,Value,Less>::iterator::operator!=
130:         (const iterator& that) const {
131:     return this->where != that.where;
132: }
133:
```

```
1: // $Id: util.tcc,v 1.3 2014-06-27 17:49:07-07 - - $
2:
3: #include <sstream>
4: #include <typeinfo>
5: using namespace std;
6:
7: template <typename item_t>
8: ostream& operator<< (ostream& out, const list<item_t>& vec) {
9:     bool want_space = false;
10:    for (const auto& item: vec) {
11:        if (want_space) cout << " ";
12:        cout << item;
13:        want_space = true;
14:    }
15:    return out;
16: }
17:
18: template <typename Type>
19: string to_string (const Type& that) {
20:     ostringstream stream;
21:     stream << that;
22:     return stream.str();
23: }
24:
25: template <typename Type>
26: Type from_string (const string& that) {
27:     stringstream stream;
28:     stream << that;
29:     Type result;
30:     if (not (stream >> result and stream.eof())) {
31:         throw domain_error (string (typeid (Type).name())
32:                               + " from_string (" + that + ")");
33:     }
34:     return result;
35: }
36:
```

```
1: // $Id: trace.cpp,v 1.3 2014-05-29 19:20:19-07 - - $
2:
3: #include <climits>
4: #include <iostream>
5: #include <limits>
6: #include <vector>
7:
8: using namespace std;
9:
10: #include "trace.h"
11:
12: using boolvec = vector<char>;
13: boolvec traceflags::flags (UCHAR_MAX + 1, false);
14: const boolvec trueflags (UCHAR_MAX + 1, true);
15:
16: void traceflags::setflags (const string& optflags) {
17:     for (char flag: optflags) {
18:         if (flag == '@') {
19:             flags = trueflags;
20:         } else {
21:             flags[flag] = true;
22:         }
23:     }
24:     // Note that TRACE can trace setflags.
25:     TRACE ('t', "optflags = " << optflags);
26: }
27:
28: //
29: // getflag -
30: //     Check to see if a certain flag is on.
31: //
32:
33: bool traceflags::getflag (char flag) {
34:     // Bug alert:
35:     // Don't TRACE this function or the stack will blow up.
36:     bool result = flags[flag];
37:     return result;
38: }
39:
```

```
1: // $Id: util.cpp,v 1.6 2015-02-11 13:04:52-08 - - $
2:
3: #include <cerrno>
4: #include <cstdlib>
5: #include <cstring>
6: #include <ctime>
7: #include <stdexcept>
8: #include <string>
9: using namespace std;
10:
11: #include "util.h"
12:
13: int sys_info::exit_status = EXIT_SUCCESS;
14: string sys_info::execname; // Must be initialized from main().
15:
16: void sys_info_error (const string& condition) {
17:     throw logic_error ("main() has " + condition
18:         + " called sys_info::set_execname()");
19: }
20:
21: void sys_info::set_execname (const string& argv0) {
22:     if (execname == "") sys_info_error ("already");
23:     int slashpos = argv0.find_last_of ('/') + 1;
24:     execname = argv0.substr (slashpos);
25:     cout << boolalpha;
26:     cerr << boolalpha;
27:     TRACE ('u', "execname = " << execname);
28: }
29:
30: const string& sys_info::get_execname () {
31:     if (execname == "") sys_info_error ("not yet");
32:     return execname;
33: }
34:
35: void sys_info::set_exit_status (int status) {
36:     if (execname == "") sys_info_error ("not yet");
37:     exit_status = status;
38: }
39:
40: int sys_info::get_exit_status () {
41:     if (execname == "") sys_info_error ("not yet");
42:     return exit_status;
43: }
44:
45: const string datestring () {
46:     time_t clock = time (nullptr);
47:     struct tm *tm_ptr = localtime (&clock);
48:     char timebuf[256];
49:     strftime (timebuf, sizeof timebuf,
50:         "%a %b %e %H:%M:%S %Z %Y", tm_ptr);
51:     return timebuf;
52: }
53:
```

```
54:
55: list<string> split (const string& line, const string& delimiters) {
56:     list<string> words;
57:     size_t end = 0;
58:     // Loop over the string, splitting out words, and for each word
59:     // thus found, append it to the output list<string>.
60:     for (;;) {
61:         size_t start = line.find_first_not_of (delimiters, end);
62:         if (start == string::npos) break;
63:         end = line.find_first_of (delimiters, start);
64:         words.push_back (line.substr (start, end - start));
65:     }
66:     TRACE ('u', words);
67:     return words;
68: }
69:
70: ostream& complain() {
71:     sys_info::set_exit_status (EXIT_FAILURE);
72:     cerr << sys_info::get_execname () << ": ";
73:     return cerr;
74: }
75:
76: void syscall_error (const string& object) {
77:     complain() << object << ": " << strerror (errno) << endl;
78: }
79:
```

```
1: // $Id: main.cpp,v 1.8 2015-04-28 19:23:13-07 - - $
2:
3: #include <cstdlib>
4: #include <exception>
5: #include <iostream>
6: #include <string>
7: #include <unistd.h>
8:
9: using namespace std;
10:
11: #include "listmap.h"
12: #include "xpair.h"
13: #include "util.h"
14:
15: using str_str_map = listmap<string,string>;
16: using str_str_pair = str_str_map::value_type;
17:
18: void scan_options (int argc, char** argv) {
19:     opterr = 0;
20:     for (;;) {
21:         int option = getopt (argc, argv, "@:");
22:         if (option == EOF) break;
23:         switch (option) {
24:             case '@':
25:                 traceflags::setflags (optarg);
26:                 break;
27:             default:
28:                 complain() << "-" << (char) optopt << ": invalid option"
29:                     << endl;
30:                 break;
31:         }
32:     }
33: }
34:
35: int main (int argc, char** argv) {
36:     sys_info::set_execname (argv[0]);
37:     scan_options (argc, argv);
38:
39:     str_str_map test;
40:     for (char** argp = &argv[optind]; argp != &argv[argc]; ++argp) {
41:         str_str_pair pair (*argp, to_string<int> (argp - argv));
42:         cout << "Before insert: " << pair << endl;
43:         test.insert (pair);
44:     }
45:
46:     for (str_str_map::iterator itor = test.begin();
47:         itor != test.end(); ++itor) {
48:         cout << "During iteration: " << *itor << endl;
49:     }
50:
51:     str_str_map::iterator itor = test.begin();
52:     test.erase (itor);
53:
54:     cout << "EXIT_SUCCESS" << endl;
55:     return EXIT_SUCCESS;
56: }
57:
```



```
1: # $Id: Makefile,v 1.10 2015-01-30 14:30:57-08 - - $
2:
3: # By default, gmake sets the variable ${CXX}=g++ on a Linux system.
4: # To use Sun Studio C++ (CC), set the variable CXX=CC, either on
5: # the command line when calling make, or in an environment variable.
6:
7: MKFILE      = Makefile
8: DEFILE      = ${MKFILE}.dep
9: NOINCL      = ci clean spotless
10: NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
11: GMAKE       = ${MAKE} --no-print-directory
12:
13: COMPILECPP  = g++ -g -O0 -Wall -Wextra -std=gnu++11
14: MAKEDEPCPP  = g++ -MM
15:
16: CPPHEADER   = listmap.h trace.h util.h xless.h xpair.h
17: TEMPLATES   = listmap.tcc util.tcc
18: CPPSOURCE   = trace.cpp util.cpp main.cpp
19: ALLCPPSRC   = ${CPPHEADER} ${TEMPLATES} ${CPPSOURCE}
20: OBJECTS     = ${CPPSOURCE:.cpp=.o}
21: EXECBIN     = keyvalue
22: OTHERS      = ${MKFILE}
23: ALLSOURCES  = ${ALLCPPSRC} ${OTHERS}
24: LISTING     = Listing.ps
25:
26: all : ${EXECBIN}
27:
28: ${EXECBIN} : ${OBJECTS}
29:             ${COMPILECPP} -o $@ ${OBJECTS}
30:
31: %.o : %.cpp
32:             ${COMPILECPP} -c $<
33:
34: ci : ${ALLSOURCES}
35:     - checksource ${ALLSOURCES}
36:     cid + ${ALLSOURCES}
37:
38: lis : ${ALLSOURCES}
39:     mkpspdf ${LISTING} ${ALLSOURCES} ${DEFILE}
40:
41: clean :
42:     - rm ${OBJECTS} ${DEFILE} core
43:
44: spotless : clean
45:     - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
46:
47: dep : ${ALLCPPSRC}
48:     @ echo "# ${DEFILE} created `LC_TIME=C date`" >${DEFILE}
49:     ${MAKEDEPCPP} ${CPPSOURCE} >>${DEFILE}
50:
51: ${DEFILE} :
52:     @ touch ${DEFILE}
53:     ${GMAKE} dep
54:
55: again :
56:     ${GMAKE} spotless dep ci all lis
57:
58: ifeq (${NEEDINCL}, )
```

04/28/15
19:23:20

\$cmpps109-wm/Assignments/asg3-listmap-templates/code/
Makefile

2/2

```
59: include ${DEPFILE}
60: endif
61:
```

```
1: # Makefile.dep created Tue Apr 28 19:23:20 PDT 2015
2: trace.o: trace.cpp trace.h
3: util.o: util.cpp util.h trace.h util.tcc
4: main.o: main.cpp listmap.h xless.h xpair.h listmap.tcc trace.h util.h \
5:  util.tcc
```