```
 1: // $Id: listmap.h,v 1.13 2017-07-17 15:16:35-07 - - $
 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&);
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>
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: listmap.tcc,v 1.11 2018-01-25 14:19:14-08 - - $
 2:
 3: #include "listmap.h"
 4: #include "debug.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:    DEBUGF ('l', reinterpret_cast<const 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:    DEBUGF ('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) {
52:    DEBUGF ('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:    DEBUGF ('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:    DEBUGF ('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:    DEBUGF ('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:    DEBUGF ('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:    DEBUGF ('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: 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: debug.h,v 1.1 2018-01-25 14:09:09-08 - - $
 2:
 3: #ifndef __DEBUG_H__
 4: #define __DEBUG_H__
 5:
 6: #include <bitset>
 7: #include <climits>
 8: #include <string>
 9: using namespace std;
10:
11: // debug -
12: //    static class for maintaining global debug flags, each indicated
13: //    by a single character.
14: // setflags -
15: //    Takes a string argument, and sets a flag for each char in the
16: //    string.  As a special case, '@', sets all flags.
17: // getflag -
18: //    Used by the DEBUGF macro to check to see if a flag has been set.
19: //    Not to be called by user code.
20:
21: class debugflags {
22:    private:
23:       using flagset = bitset<UCHAR_MAX + 1>;
24:       static flagset flags;
25:    public:
26:       static void setflags (const string& optflags);
27:       static bool getflag (char flag);
28:       static void where (char flag, const char* file, int line,
29:                          const char* pretty_function);
30: };
31:
```

```
32:
33: // DEBUGF -
34: //    Macro which expands into debug code.  First argument is a
35: //    debug flag char, second argument is output code that can
36: //    be sandwiched between <<.  Beware of operator precedence.
37: //    Example:
38: //       DEBUGF ('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: #ifdef NDEBUG
43: #define DEBUGF(FLAG,CODE) ;
44: #define DEBUGS(FLAG,STMT) ;
45: #else
46: #define DEBUGF(FLAG,CODE) { \
47:            if (debugflags::getflag (FLAG)) { \
48:               debugflags::where (FLAG, __FILE__, __LINE__, \
49:                                  __PRETTY_FUNCTION__); \
50:               cerr << CODE << endl; \
51:            } \
52:         }
53: #define DEBUGS(FLAG,STMT) { \
54:            if (debugflags::getflag (FLAG)) { \
55:               debugflags::where (FLAG, __FILE__, __LINE__, \
56:                                  __PRETTY_FUNCTION__); \
57:               STMT; \
58:            } \
59:         }
60: #endif
61:
62: #endif
63:
```

```
 1: // $Id: debug.cpp,v 1.2 2018-01-25 14:12:59-08 - - $
 2:
 3: #include <climits>
 4: #include <iostream>
 5: #include <vector>
 6:
 7: using namespace std;
 8:
 9: #include "debug.h"
10: #include "util.h"
11:
12: debugflags::flagset debugflags::flags {};
13:
14: void debugflags::setflags (const string& initflags) {
15:    for (const unsigned char flag: initflags) {
16:       if (flag == '@') flags.set();
17:                   else flags.set (flag, true);
18:    }
19: }
20:
21: // getflag -
22: //    Check to see if a certain flag is on.
23:
24: bool debugflags::getflag (char flag) {
25:    // WARNING: Don't TRACE this function or the stack will blow up.
26:    return flags.test (static_cast<unsigned char> (flag));
27: }
28:
29: void debugflags::where (char flag, const char* file, int line,
30:                         const char* pretty_function) {
31:    cout << sys_info::execname() << ": DEBUG(" << flag << ") "
32:         << file << "[" << line << "] " << endl
33:         << "   " << pretty_function << endl;
34: }
35:
```

```
 1: // $Id: util.h,v 1.6 2018-01-25 14:18:43-08 - - $
 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: using namespace std;
17:
18: //
19: // sys_info -
20: //    Keep track of execname and exit status.  Must be initialized
21: //    as the first thing done inside main.  Main should call:
22: //       sys_info::set_execname (argv[0]);
23: //    before anything else.
24: //
25:
26: class sys_info {
27:    private:
28:       static string execname_;
29:       static int exit_status_;
30:       static void execname (const string& argv0);
31:       friend int main (int argc, char** argv);
32:    public:
33:       static const string& execname ();
34:       static void exit_status (int status);
35:       static int exit_status ();
36: };
37:
38: //
39: // datestring -
40: //    Return the current date, as printed by date(1).
41: //
42:
43: const string datestring ();
44:
45: //
46: // split -
47: //    Split a string into a list<string>..  Any sequence
48: //    of chars in the delimiter string is used as a separator.  To
49: //    Split a pathname, use "/".  To split a shell command, use " ".
50: //
51:
52: list<string> split (const string& line, const string& delimiter);
53:
```

```
 54:
 55: //
 56: // complain -
 57: //    Used for starting error messages.  Sets the exit status to
 58: //    EXIT_FAILURE, writes the program name to cerr, and then
 59: //    returns the cerr ostream.  Example:
 60: //       complain() << filename << ": some problem" << endl;
 61: //
 62:
 63: ostream& complain();
 64:
 65: //
 66: // syscall_error -
 67: //    Complain about a failed system call.  Argument is the name
 68: //    of the object causing trouble.  The extern errno must contain
 69: //    the reason for the problem.
 70: //
 71:
 72: void syscall_error (const string&);
 73:
 74: //
 75: // operator<< (list) -
 76: //    An overloaded template operator which allows lists to be
 77: //    printed out as a single operator, each element separated from
 78: //    the next with spaces.  The item_t must have an output operator
 79: //    defined for it.
 80: //
 81:
 82: template <typename item_t>
 83: ostream& operator<< (ostream& out, const list<item_t>& vec);
 84:
 85: //
 86: // string to_string (thing) -
 87: //    Convert anything into a string if it has an ostream<< operator.
 88: //
 89:
 90: template <typename item_t>
 91: string to_string (const item_t&);
 92:
 93: //
 94: // thing from_string (cons string&) -
 95: //    Scan a string for something if it has an istream>> operator.
 96: //
 97:
 98: template <typename item_t>
 99: item_t from_string (const string&);
100:
101: //
102: // Put the RCS Id string in the object file.
103: //
104:
105: #include "util.tcc"
106: #endif
107:
```

```
 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: util.cpp,v 1.14 2018-01-25 14:18:43-08 - - $
 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 "debug.h"
12: #include "util.h"
13:
14: int sys_info::exit_status_ = EXIT_SUCCESS;
15: string sys_info::execname_; // Must be initialized from main().
16:
17: void sys_info_error (const string& condition) {
18:    throw logic_error ("main() has " + condition
19:                       + " called sys_info::execname()");
20: }
21:
22: void sys_info::execname (const string& argv0) {
23:    if (execname_ != "") sys_info_error ("already");
24:    int slashpos = argv0.find_last_of ('/') + 1;
25:    execname_ = argv0.substr (slashpos);
26:    cout << boolalpha;
27:    cerr << boolalpha;
28:    DEBUGF ('u', "execname_ = " << execname_);
29: }
30:
31: const string& sys_info::execname () {
32:    if (execname_ == "") sys_info_error ("not yet");
33:    return execname_;
34: }
35:
36: void sys_info::exit_status (int status) {
37:    if (execname_ == "") sys_info_error ("not yet");
38:    exit_status_ = status;
39: }
40:
41: int sys_info::exit_status () {
42:    if (execname_ == "") sys_info_error ("not yet");
43:    return exit_status_;
44: }
45:
46: const string datestring () {
47:    time_t clock = time (nullptr);
48:    struct tm *tm_ptr = localtime (&clock);
49:    char timebuf[256];
50:    strftime (timebuf, sizeof timebuf,
51:              "%a %b %e %H:%M:%S %Z %Y", tm_ptr);
52:    return timebuf;
53: }
54:
```

```
55:
56: list<string> split (const string& line, const string& delimiters) {
57:    list<string> words;
58:    size_t end = 0;
59:    // Loop over the string, splitting out words, and for each word
60:    // thus found, append it to the output list<string>.
61:    for (;;) {
62:       size_t start = line.find_first_not_of (delimiters, end);
63:       if (start == string::npos) break;
64:       end = line.find_first_of (delimiters, start);
65:       words.push_back (line.substr (start, end - start));
66:    }
67:    DEBUGF ('u', words);
68:    return words;
69: }
70:
71: ostream& complain() {
72:    sys_info::exit_status (EXIT_FAILURE);
73:    cerr << sys_info::execname () << ": ";
74:    return cerr;
75: }
76:
77: void syscall_error (const string& object) {
78:    complain() << object << ": " << strerror (errno) << endl;
79: }
80:
```

```
 1: // $Id: main.cpp,v 1.11 2018-01-25 14:19:29-08 - - $
 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:             debugflags::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::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 1.20 2018-01-31 18:30:15-08 - - $
 2:
 3: MKFILE      = Makefile
 4: DEPFILE     = ${MKFILE}.dep
 5: NOINCL      = ci clean spotless
 6: NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
 7: GMAKE       = ${MAKE} --no-print-directory
 8:
 9: COMPILECPP  = g++ -std=gnu++17 -g -O0 -Wall -Wextra -Wold-style-cast
10: MAKEDEPCPP  = g++ -std=gnu++17 -MM
11:
12: MODULES     = listmap xless xpair debug util main
13: CPPSOURCE   = ${wildcard ${MODULES:=.cpp}}
14: OBJECTS     = ${CPPSOURCE:.cpp=.o}
15: SOURCELIST  = ${foreach MOD, ${MODULES}, ${MOD}.h ${MOD}.tcc ${MOD}.cpp}
16: ALLSOURCE   = ${wildcard ${SOURCELIST}}
17: EXECBIN     = keyvalue
18: OTHERS      = ${MKFILE} ${DEPFILE}
19: ALLSOURCES  = ${ALLSOURCE} ${OTHERS}
20: LISTING     = Listing.ps
21:
22: all : ${EXECBIN}
23:
24: ${EXECBIN} : ${OBJECTS}
25:         ${COMPILECPP} -o $@ ${OBJECTS}
26:
27: %.o : %.cpp
28:         checksource $<
29:         cpplint.py.perl $<
30:         ${COMPILECPP} -c $<
31:
32: ci : ${ALLSOURCES}
33:         cid + ${ALLSOURCES}
34:
35: lis : ${ALLSOURCES}
36:         mkpspdf ${LISTING} ${ALLSOURCES}
37:
38: clean :
39:         - rm ${OBJECTS} ${DEPFILE} core
40:
41: spotless : clean
42:         - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
43:
44: dep : ${ALLCPPSRC}
45:         @ echo "# ${DEPFILE} created `LC_TIME=C date`" >${DEPFILE}
46:         ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPFILE}
47:
48: ${DEPFILE} :
49:         @ touch ${DEPFILE}
50:         ${GMAKE} dep
51:
52: again :
53:         ${GMAKE} spotless dep ci all lis
54:
55: ifeq (${NEEDINCL}, )
56: include ${DEPFILE}
57: endif
58:
```

```
1: # Makefile.dep created Wed Jan 31 18:30:14 PST 2018
2: debug.o: debug.cpp debug.h util.h util.tcc
3: util.o: util.cpp debug.h util.h util.tcc
4: main.o: main.cpp listmap.h xless.h xpair.h listmap.tcc debug.h util.h \
5:  util.tcc
```