```
 1: // $Id: listmap.h,v 1.21 2019-10-30 12:44:53-07 - - $
 2:
 3: #ifndef __LISTMAP_H__
 4: #define __LISTMAP_H__
 5:
 6: #include "xless.h"
 7: #include "xpair.h"
 8:
 9: template <typename key_t, typename mapped_t, class less_t=xless<key_t>>
10: class listmap {
11:    public:
12:       using key_type = key_t;
13:       using mapped_type = mapped_t;
14:       using value_type = xpair<const key_type, mapped_type>;
15:    private:
16:       less_t 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& value_):
26:                link (next_, prev_), value(value_){}
27:       };
28:       node* anchor() { return static_cast<node*> (&anchor_); }
29:       link anchor_ {anchor(), anchor()};
30:    public:
31:       class iterator;
32:       listmap(){};
33:       listmap (const listmap&);
34:       listmap& operator= (const listmap&);
35:       ~listmap();
36:       iterator insert (const value_type&);
37:       iterator find (const key_type&);
38:       iterator erase (iterator position);
39:       iterator begin() { return anchor()->next; }
40:       iterator end() { return anchor(); }
41:       bool empty() const { return anchor_->next == anchor_; }
42: };
43:
```

```
44:
45: template <typename key_t, typename mapped_t, class less_t>
46: class listmap<key_t,mapped_t,less_t>::iterator {
47:    private:
48:       friend class listmap<key_t,mapped_t,less_t>;
49:       listmap<key_t,mapped_t,less_t>::node* where {nullptr};
50:       iterator (node* where_): where(where_){};
51:    public:
52:       iterator() {}
53:       value_type& operator*() { return where->value; }
54:       value_type* operator->() { return &(where->value); }
55:       iterator& operator++() { where = where->next; return *this; }
56:       iterator& operator--() { where = where->prev; return *this; }
57:       void erase();
58:       bool operator== (const iterator& that) const {
59:          return this->where == that.where;
60:       }
61:       bool operator!= (const iterator& that) const {
62:          return this->where != that.where;
63:       }
64: };
65:
66: #include "listmap.tcc"
67: #endif
68:
```

```
 1: // $Id: listmap.tcc,v 1.15 2019-10-30 12:44:53-07 - - $
 2:
 3: #include "listmap.h"
 4: #include "debug.h"
 5:
 6: //
 7: //////////////////////////////////////////////////////////////////
 8: // Operations on listmap.
 9: //////////////////////////////////////////////////////////////////
10: //
11:
12: //
13: // listmap::~listmap()
14: //
15: template <typename key_t, typename mapped_t, class less_t>
16: listmap<key_t,mapped_t,less_t>::~listmap() {
17:    DEBUGF ('l', reinterpret_cast<const void*> (this));
18: }
19:
20: //
21: // iterator listmap::insert (const value_type&)
22: //
23: template <typename key_t, typename mapped_t, class less_t>
24: typename listmap<key_t,mapped_t,less_t>::iterator
25: listmap<key_t,mapped_t,less_t>::insert (const value_type& pair) {
26:    DEBUGF ('l', &pair << "->" << pair);
27:    return iterator();
28: }
29:
30: //
31: // listmap::find(const key_type&)
32: //
33: template <typename key_t, typename mapped_t, class less_t>
34: typename listmap<key_t,mapped_t,less_t>::iterator
35: listmap<key_t,mapped_t,less_t>::find (const key_type& that) {
36:    DEBUGF ('l', that);
37:    return iterator();
38: }
39:
40: //
41: // iterator listmap::erase (iterator position)
42: //
43: template <typename key_t, typename mapped_t, class less_t>
44: typename listmap<key_t,mapped_t,less_t>::iterator
45: listmap<key_t,mapped_t,less_t>::erase (iterator position) {
46:    DEBUGF ('l', &*position);
47:    return iterator();
48: }
49:
50:
```

```
 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 1.5 2019-02-21 17:27:16-08 - - $
 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_t and second_t do so with their default ctors.
17: //
18:
19: template <typename first_t, typename second_t>
20: struct xpair {
21:    first_t first{};
22:    second_t second{};
23:    xpair(){}
24:    xpair (const first_t& first_, const second_t& second_):
25:                  first(first_), second(second_) {}
26: };
27:
28: template <typename first_t, typename second_t>
29: ostream& operator<< (ostream& out,
30:                      const xpair<first_t,second_t>& pair) {
31:    out << "{" << pair.first << "," << pair.second << "}";
32:    return out;
33: }
34:
35: #endif
36:
```

```
 1: // $Id: debug.h,v 1.2 2019-10-22 12:41:48-07 - - $
 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.
13: // setflags -
14: //    Takes a string argument, and sets a flag for each char in the
15: //    string.  As a special case, '@', sets all flags.
16: // getflag -
17: //    Used by the DEBUGF macro to check to see if a flag has been set.
18: //    Not to be called by user code.
19:
20: class debugflags {
21:    private:
22:       using flagset = bitset<UCHAR_MAX + 1>;
23:       static flagset flags;
24:    public:
25:       static void setflags (const string& optflags);
26:       static bool getflag (char flag);
27:       static void where (char flag, const char* file, int line,
28:                          const char* pretty_function);
29: };
30:
```

```
31:
32: // DEBUGF -
33: //    Macro which expands into debug code.  First argument is a
34: //    debug flag char, second argument is output code that can
35: //    be sandwiched between <<.  Beware of operator precedence.
36: //    Example:
37: //       DEBUGF ('u', "foo = " << foo);
38: //    will print two words and a newline if flag 'u' is  on.
39: //    Traces are preceded by filename, line number, and function.
40:
41: #ifdef NDEBUG
42: #define DEBUGF(FLAG,CODE) ;
43: #define DEBUGS(FLAG,STMT) ;
44: #else
45: #define DEBUGF(FLAG,CODE) { \
46:            if (debugflags::getflag (FLAG)) { \
47:               debugflags::where (FLAG, __FILE__, __LINE__, \
48:                                  __PRETTY_FUNCTION__); \
49:               cerr << CODE << endl; \
50:            } \
51:         }
52: #define DEBUGS(FLAG,STMT) { \
53:            if (debugflags::getflag (FLAG)) { \
54:               debugflags::where (FLAG, __FILE__, __LINE__, \
55:                                  __PRETTY_FUNCTION__); \
56:               STMT; \
57:            } \
58:         }
59: #endif
60:
61: #endif
62:
```

```
 1: // $Id: debug.cpp,v 1.3 2019-10-22 12:41:48-07 - - $
 2:
 3: #include <climits>
 4: #include <iostream>
 5: using namespace std;
 6:
 7: #include "debug.h"
 8: #include "util.h"
 9:
10: debugflags::flagset debugflags::flags {};
11:
12: void debugflags::setflags (const string& initflags) {
13:    for (const unsigned char flag: initflags) {
14:       if (flag == '@') flags.set();
15:                   else flags.set (flag, true);
16:    }
17: }
18:
19: // getflag -
20: //    Check to see if a certain flag is on.
21:
22: bool debugflags::getflag (char flag) {
23:    // WARNING: Don't TRACE this function or the stack will blow up.
24:    return flags.test (static_cast<unsigned char> (flag));
25: }
26:
27: void debugflags::where (char flag, const char* file, int line,
28:                         const char* pretty_function) {
29:    cout << sys_info::execname() << ": DEBUG(" << flag << ") "
30:         << file << "[" << line << "] " << endl
31:         << "   " << pretty_function << endl;
32: }
33:
```

```
 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:
```

```
# $Id: Makefile,v 1.24 2019-10-22 12:41:48-07 - - $

MKFILE      = Makefile
DEPFILE     = ${MKFILE}.dep
NOINCL      = ci clean spotless
NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
GMAKE       = ${MAKE} --no-print-directory

GPPWARN     = -Wall -Wextra -Wpedantic -Wshadow -Wold-style-cast
GPPOPTS     = ${GPPWARN} -fdiagnostics-color=never
COMPILECPP  = g++ -std=gnu++17 -g -O0 ${GPPOPTS}
MAKEDEPCPP  = g++ -std=gnu++17 -MM ${GPPOPTS}
UTILBIN     = /afs/cats.ucsc.edu/courses/cse111-wm/bin

MODULES     = listmap xless xpair debug util main
CPPSOURCE   = ${wildcard ${MODULES:=.cpp}}
OBJECTS     = ${CPPSOURCE:.cpp=.o}
SOURCELIST  = ${foreach MOD, ${MODULES}, ${MOD}.h ${MOD}.tcc ${MOD}.cpp}
ALLSOURCE   = ${wildcard ${SOURCELIST}}
EXECBIN     = keyvalue
OTHERS      = ${MKFILE} ${DEPFILE}
ALLSOURCES  = ${ALLSOURCE} ${OTHERS}
LISTING     = Listing.ps

all : ${EXECBIN}

${EXECBIN} : ${OBJECTS}
        ${COMPILECPP} -o $@ ${OBJECTS}

%.o : %.cpp
        - ${UTILBIN}/checksource $<
        - ${UTILBIN}/cpplint.py.perl $<
        ${COMPILECPP} -c $<

ci : ${ALLSOURCES}
        ${UTILBIN}/cid + ${ALLSOURCES}
        - ${UTILBIN}/checksource ${ALLSOURCES}

lis : ${ALLSOURCES}
        mkpspdf ${LISTING} ${ALLSOURCES}

clean :
        - rm ${OBJECTS} ${DEPFILE} core

spotless : clean
        - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}

dep : ${ALLCPPSRC}
        @ echo "# ${DEPFILE} created `LC_TIME=C date`" >${DEPFILE}
        ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPFILE}

${DEPFILE} :
        @ touch ${DEPFILE}
        ${GMAKE} dep

again :
        ${GMAKE} spotless dep ci all lis
```

```
59: ifeq (${NEEDINCL}, )
60: include ${DEPFILE}
61: endif
62:
```

```
1: # Makefile.dep created Fri Jan 31 13:20:40 PST 2020
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
```