

```
1: // $Id: listmap.h,v 1.28 2021-02-22 12:17:11-08 - - $
2:
3: #ifndef __LISTMAP_H__
4: #define __LISTMAP_H__
5:
6: #include "debug.h"
7: #include "xless.h"
8: #include "xpair.h"
9:
10: #define SHOW_LINK(FLAG,PTR) { \
11:     DEBUGB (FLAG, #PTR << "=" << PTR \
12:         << ": next=" << PTR->next \
13:         << ", prev=" << PTR->prev); \
14: }
15:
16: template <typename key_t, typename mapped_t, class less_t=xless<key_t>>
17: class listmap {
18:     public:
19:         using key_type = key_t;
20:         using mapped_type = mapped_t;
21:         using value_type = xpair<const key_type, mapped_type>;
22:     private:
23:         less_t less;
24:         struct node;
25:         struct link {
26:             node* next{};
27:             node* prev{};
28:             link (node* next_, node* prev_): next(next_), prev(prev_){}
29:         };
30:         struct node: link {
31:             value_type value{};
32:             node (node* next_, node* prev_, const value_type& value_):
33:                 link (next_, prev_), value(value_){}
34:         };
35:         node* anchor() { return static_cast<node*> (&anchor_); }
36:         link anchor_ {anchor(), anchor()};
37:     public:
38:         class iterator;
39:         listmap(){};
40:         listmap (const listmap&);
41:         listmap& operator= (const listmap&);
42:         ~listmap();
43:         iterator insert (const value_type&);
44:         iterator find (const key_type&);
45:         iterator erase (iterator position);
46:         iterator begin() { return anchor()->next; }
47:         iterator end() { return anchor(); }
48:         bool empty() const { return anchor_.next == &anchor_; }
49:         operator bool() const { return not empty(); }
50: };
51:
```

```
52:
53: template <typename key_t, typename mapped_t, class less_t>
54: class listmap<key_t,mapped_t,less_t>::iterator {
55:     friend class listmap<key_t,mapped_t,less_t>;
56:     private:
57:         listmap<key_t,mapped_t,less_t>::node* where {nullptr};
58:         iterator (node* where_): where(where_){};
59:     public:
60:         iterator() {}
61:         value_type& operator*() {
62:             SHOW_LINK ('b', where);
63:             return where->value;
64:         }
65:         value_type* operator->() { return &(where->value); }
66:         iterator& operator++() { where = where->next; return *this; }
67:         iterator& operator--() { where = where->prev; return *this; }
68:         bool operator== (const iterator& that) const {
69:             return this->where == that.where;
70:         }
71:         bool operator!= (const iterator& that) const {
72:             return this->where != that.where;
73:         }
74:         operator bool() const { return where != nullptr; }
75: };
76:
77: #include "listmap.tcc"
78: #endif
79:
```

```
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 1.5 2020-11-12 18:37:41-08 - - $
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.
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 DEBUGB(FLAG, CODE) ;
43: #define DEBUGF(FLAG, CODE) ;
44: #define DEBUGS(FLAG, STMT) ;
45: #else
46: #define DEBUGB(FLAG, CODE) { \
47:     if (debugflags::getflag (FLAG)) { \
48:         debugflags::where (FLAG, __FILE__, __LINE__, \
49:             __PRETTY_FUNCTION__); \
50:         cerr << CODE << endl; \
51:     } \
52: }
53: #define DEBUGF(FLAG, CODE) { \
54:     if (debugflags::getflag (FLAG)) { \
55:         debugflags::where (FLAG, __FILE__, __LINE__, \
56:             __PRETTY_FUNCTION__); \
57:         cerr << CODE << endl; \
58:     } \
59: }
60: #define DEBUGS(FLAG, STMT) { \
61:     if (debugflags::getflag (FLAG)) { \
62:         debugflags::where (FLAG, __FILE__, __LINE__, \
63:             __PRETTY_FUNCTION__); \
64:         STMT; \
65:     } \
66: }
67: #endif
68:
69: #endif
70:
```

```
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 1.9 2021-04-28 12:12:32-07 - - $
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 <stdexcept>
14: #include <string>
15: using namespace std;
16:
17: //
18: // sys_info -
19: //     Keep track of execname and exit status.  Must be initialized
20: //     as the first thing done inside main.  Main should call:
21: //         sys_info::set_execname (argv[0]);
22: //     before anything else.
23: //
24:
25: class sys_info {
26:     private:
27:         static string execname_;
28:         static int exit_status_;
29:         static void execname (const string& argv0);
30:         friend int main (int, char**);
31:     public:
32:         static const string& execname ();
33:         static void exit_status (int status);
34:         static int exit_status ();
35: };
36:
```

```
37:
38: //
39: // complain -
40: //     Used for starting error messages.  Sets the exit status to
41: //     EXIT_FAILURE, writes the program name to cerr, and then
42: //     returns the cerr ostream.  Example:
43: //         complain() << filename << ": some problem" << endl;
44: //
45:
46: ostream& complain();
47:
48: //
49: // syscall_error -
50: //     Complain about a failed system call.  Argument is the name
51: //     of the object causing trouble.  The extern errno must contain
52: //     the reason for the problem.
53: //
54:
55: void syscall_error (const string&);
56:
57: //
58: // string to_string (thing) -
59: //     Convert anything into a string if it has an ostream<< operator.
60: //
61:
62: template <typename item_t>
63: string to_string (const item_t&);
64:
65: //
66: // thing from_string (const string&) -
67: //     Scan a string for something if it has an istream>> operator.
68: //
69:
70: template <typename item_t>
71: item_t from_string (const string&);
72:
73: //
74: // Put the RCS Id string in the object file.
75: //
76:
77: #include "util.tcc"
78: #endif
79:
```

```
1: // $Id: util.tcc,v 1.4 2020-02-06 12:33:29-08 - - $
2:
3: #include <sstream>
4: #include <typeinfo>
5: using namespace std;
6:
7: template <typename Type>
8: string to_string (const Type& that) {
9:     ostringstream stream;
10:    stream << that;
11:    return stream.str();
12: }
13:
14: template <typename Type>
15: Type from_string (const string& that) {
16:    stringstream stream;
17:    stream << that;
18:    Type result;
19:    if (not (stream >> result and stream.eof())) {
20:        throw domain_error (string (typeid (Type).name())
21:                               + " from_string (" + that + ")");
22:    }
23:    return result;
24: }
25:
```

```
1: // $Id: util.cpp,v 1.18 2020-02-06 12:55:59-08 - - $
2:
3: #include <cassert>
4: #include <cerrno>
5: #include <cstdlib>
6: #include <cstring>
7: #include <ctime>
8: #include <stdexcept>
9: #include <string>
10: using namespace std;
11:
12: #include "debug.h"
13: #include "util.h"
14:
15: int sys_info::exit_status_ = EXIT_SUCCESS;
16: string sys_info::execname_; // Must be initialized from main().
17:
18: void sys_info::execname (const string& argv0) {
19:     assert (execname_ == "");
20:     int slashpos = argv0.find_last_of ('/') + 1;
21:     execname_ = argv0.substr (slashpos);
22:     cout << boolalpha;
23:     cerr << boolalpha;
24:     DEBUGF ('u', "execname_ = " << execname_);
25: }
26:
27: const string& sys_info::execname () {
28:     assert (execname_ != "");
29:     return execname_;
30: }
31:
32: void sys_info::exit_status (int status) {
33:     assert (execname_ != "");
34:     exit_status_ = status;
35: }
36:
37: int sys_info::exit_status () {
38:     assert (execname_ != "");
39:     return exit_status_;
40: }
41:
42: ostream& complain() {
43:     sys_info::exit_status (EXIT_FAILURE);
44:     cerr << sys_info::execname () << ": ";
45:     return cerr;
46: }
47:
48: void syscall_error (const string& object) {
49:     complain() << object << ": " << strerror (errno) << endl;
50: }
51:
```

```
1: // $Id: main.cpp,v 1.13 2021-02-01 18:58:18-08 - - $
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:     cout << test << endl;
41:     for (char** argp = &argv[optind]; argp != &argv[argc]; ++argp) {
42:         str_str_pair pair (*argp, to_string<int> (argp - argv));
43:         cout << "Before insert: " << pair << endl;
44:         test.insert (pair);
45:     }
46:
47:     cout << test.empty() << endl;
48:     for (str_str_map::iterator itor = test.begin();
49:         itor != test.end(); ++itor) {
50:         cout << "During iteration: " << *itor << endl;
51:     }
52:
53:     str_str_map::iterator itor = test.begin();
54:     test.erase (itor);
55:
56:     cout << "EXIT_SUCCESS" << endl;
57:     return EXIT_SUCCESS;
58: }
```

59:

```
1: # $Id: Makefile,v 1.26 2020-10-22 16:52:39-07 - - $
2:
3: MKFILE      = Makefile
4: DEPPFILE    = ${MKFILE}.dep
5: NOINCL      = ci clean spotless check lint
6: NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
7: GMAKE       = ${MAKE} --no-print-directory
8:
9: GPPWARN      = -Wall -Wextra -Wpedantic -Wshadow -Wold-style-cast
10: GPPOPTS     = ${GPPWARN} -fdiagnostics-color=never
11: COMPILECPP  = g++ -std=gnu++17 -g -O0 ${GPPOPTS}
12: MAKEDEPCPP  = g++ -std=gnu++17 -MM ${GPPOPTS}
13: UTILBIN     = /afs/cats.ucsc.edu/courses/cse111-wm/bin
14:
15: MODULES     = listmap xless xpair debug util main
16: CPPSOURCE   = ${wildcard ${MODULES:=.cpp}}
17: OBJECTS     = ${CPPSOURCE:.cpp=.o}
18: SOURCELIST  = ${foreach MOD, ${MODULES}, ${MOD}.h ${MOD}.tcc ${MOD}.cpp}
19: ALLSOURCE   = ${wildcard ${SOURCELIST}}
20: EXECBIN     = keyvalue
21: OTHERS      = ${MKFILE} ${DEPPFILE}
22: ALLSOURCES  = ${ALLSOURCE} ${OTHERS}
23: LISTING     = Listing.ps
24:
25: all : ${EXECBIN}
26:
27: ${EXECBIN} : ${OBJECTS}
28:     ${COMPILECPP} -o $@ ${OBJECTS}
29:
30: %.o : %.cpp
31:     ${COMPILECPP} -c $<
32:
33: lint : ${CPPSOURCE}
34:     ${UTILBIN}/cpplint.py.perl ${CPPSOURCE}
35:
36: check : ${ALLSOURCES}
37:     ${UTILBIN}/checksource ${ALLSOURCES}
38:
39: ci : ${ALLSOURCES}
40:     ${UTILBIN}/cid -is ${ALLSOURCES}
41:
42: lis : ${ALLSOURCES}
43:     mkpspdf ${LISTING} ${ALLSOURCES}
44:
45: clean :
46:     - rm ${OBJECTS} ${DEPPFILE} core
47:
48: spotless : clean
49:     - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
50:
51: dep : ${ALLCPPSRC}
52:     @ echo "# ${DEPPFILE} created `LC_TIME=C date`" >${DEPPFILE}
53:     ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPPFILE}
54:
55: ${DEPPFILE} :
56:     @ touch ${DEPPFILE}
57:     ${GMAKE} dep
58:
```

```
59: again :
60:         ${GMAKE} spotless dep ci all lis
61:
62: ifeq (${NEEDINCL}, )
63: include ${DEPFILE}
64: endif
65:
```



```
1: # Makefile.dep created Wed Apr 28 12:12:32 PDT 2021
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 debug.h xless.h xpair.h listmap.tcc util.h \
5:  util.tcc
```