

```
1: // $Id: listmap.h,v 1.29 2021-05-22 02:29:26-07 - - $
2: //Kai O'Brien (kimobrie@ucsc.edu)
3:
4: #ifndef __LISTMAP_H__
5: #define __LISTMAP_H__
6:
7: #include "debug.h"
8: #include "xless.h"
9: #include "xpair.h"
10:
11: #define SHOW_LINK(FLAG,PTR) { \
12:     DEBUGB (FLAG, #PTR << "=" << PTR \
13:         << ": next=" << PTR->next \
14:         << ", prev=" << PTR->prev); \
15: }
16:
17: template <typename key_t, typename mapped_t, class less_t=xless<key_t>>
18: class listmap {
19:     public:
20:         using key_type = key_t;
21:         using mapped_type = mapped_t;
22:         using value_type = xpair<const key_type, mapped_type>;
23:     private:
24:         less_t less;
25:         struct node;
26:         struct link {
27:             node* next{};
28:             node* prev{};
29:             link (node* next_, node* prev_): next(next_), prev(prev_){}
30:         };
31:         struct node: link {
32:             value_type value{};
33:             node (node* next_, node* prev_, const value_type& value_):
34:                 link (next_, prev_), value(value_){}
35:         };
36:         node* anchor() { return static_cast<node*> (&anchor_); }
37:         link anchor_ {anchor(), anchor()};
38:     public:
39:         class iterator;
40:         listmap(){};
41:         listmap (const listmap&);
42:         listmap& operator= (const listmap&);
43:         ~listmap();
44:         iterator insert (const value_type&);
45:         iterator find (const key_type&);
46:         iterator erase (iterator position);
47:         iterator begin() { return anchor()->next; }
48:         iterator end() { return anchor(); }
49:         bool empty() const { return anchor_.next == &anchor_; }
50:         operator bool() const { return not empty(); }
51: };
52:
```

```
53:
54: template <typename key_t, typename mapped_t, class less_t>
55: class listmap<key_t,mapped_t,less_t>::iterator {
56:     friend class listmap<key_t,mapped_t,less_t>;
57:     private:
58:         listmap<key_t,mapped_t,less_t>::node* where {nullptr};
59:         iterator (node* where_): where(where_){};
60:     public:
61:         iterator() {}
62:         value_type& operator*() {
63:             SHOW_LINK ('b', where);
64:             return where->value;
65:         }
66:         value_type* operator->() { return &(where->value); }
67:         iterator& operator++() { where = where->next; return *this; }
68:         iterator& operator--() { where = where->prev; return *this; }
69:         bool operator== (const iterator& that) const {
70:             return this->where == that.where;
71:         }
72:         bool operator!= (const iterator& that) const {
73:             return this->where != that.where;
74:         }
75:         operator bool() const { return where != nullptr; }
76: };
77:
78: #include "listmap.tcc"
79: #endif
80:
```

```
1: // $Id: listmap.tcc,v 1.25 2021-05-23 00:08:49-07 - - $
2: //Kai O'Brien (kimobrie@ucsc.edu)
3:
4: #include "listmap.h"
5: #include "debug.h"
6:
7: //
8: ///////////////////////////////////////////////////////////////////
9: // Operations on listmap.
10: ///////////////////////////////////////////////////////////////////
11: //
12: //do ~ insert find and erase + main
13:
14: //
15: // listmap::~listmap()
16: //
17: template <typename key_t, typename mapped_t, class less_t>
18: listmap<key_t,mapped_t,less_t>::~listmap() {
19:     DEBUGF ('l', reinterpret_cast<const void*> (this));
20:     //typical double linked list deconstructor
21:     //DONT DELETE ANCHOR!
22:     node* temp1 = anchor()->next;
23:     node* temp2;
24:     while(temp1!=anchor()){
25:         temp2 = temp1;
26:         temp1 = temp1->next;
27:         erase(temp2);
28:     }
29: }
30:
31: //
32: // iterator listmap::insert (const value_type&)
33: //
34: template <typename key_t, typename mapped_t, class less_t>
35: typename listmap<key_t,mapped_t,less_t>::iterator
36: listmap<key_t,mapped_t,less_t>::insert (const value_type& pair) {
37:     DEBUGF ('l', &pair << "->" << pair);
38:     //if empty
39:     if(empty()){
40:         node *empty_node = new node(anchor(),anchor(),pair);
41:         anchor_.next = empty_node;
42:         anchor_.prev = empty_node;
43:         return iterator(empty_node);
44:     }
45:     //otherwise
46:     //if key is already there, the value is replaced
47:     node *new_node = nullptr;
48:     for (auto itor = begin(); itor != end(); ++itor) {
49:         //if the itr is == key, update value
50:         if(!less(itor->first,pair.first) &&
51:             !less(pair.first,itor->first)) {
52:             itor->second = pair.second;//this works
53:             return iterator(new_node);
54:         }
55:         //if pair.first is >= itor, not less than itor
56:         else if(less(pair.first,itor->first)){
57:             new_node = new node(itor.where,itor.where->prev,pair);
58:             itor.where->prev->next = new_node;
```

```
59:         itor.where->prev = new_node;
60:         return iterator(new_node);
61:     }
62:
63: }
64: //if at the end, just insert after everything
65: new_node = new node(anchor(), anchor_.prev, pair);
66: anchor_.prev->next = new_node;
67: anchor_.prev = new_node;
68: return iterator(new_node);
69:
70: }
71:
72: //
73: // listmap::find(const key_type&)
74: // cant use ==, must use less()
75: //if not is less and not is greater
76: template <typename key_t, typename mapped_t, class less_t>
77: typename listmap<key_t, mapped_t, less_t>::iterator
78: listmap<key_t, mapped_t, less_t>::find (const key_type& that) {
79:     DEBUGF ('l', that);
80:     auto itor = begin();
81:     while(itor !=end()){
82:         if(!less(itor->first,that) && !less(that,itor->first)){
83:             break;
84:         }
85:         ++itor;
86:     }
87:     return iterator(itor);
88: }
89:
90: //
91: // iterator listmap::erase (iterator position)
92: //
93: template <typename key_t, typename mapped_t, class less_t>
94: typename listmap<key_t, mapped_t, less_t>::iterator
95: listmap<key_t, mapped_t, less_t>::erase (iterator position) {
96:     DEBUGF ('l', &*position);
97:     //dont need to iterate because have .where
98:     node *temp = position.where;
99:     iterator p = temp->prev;
100:    iterator n = temp->next;
101:    p.where->next = n.where;
102:    n.where->prev = p.where;
103:
104:
105:    delete temp;
106:    return n; //should return temp->next's position
107:
108: }
109:
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 1.5 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 1.6 2021-05-22 02:29:26-07 - - $
2: //Kai O'Brien (kimobrie@ucsc.edu)
3:
4: #ifndef __DEBUG_H__
5: #define __DEBUG_H__
6:
7: #include <bitset>
8: #include <climits>
9: #include <string>
10: using namespace std;
11:
12: // debug -
13: //      static class for maintaining global debug flags.
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 DEBUGB(FLAG, CODE) ;
44: #define DEBUGF(FLAG, CODE) ;
45: #define DEBUGS(FLAG, STMT) ;
46: #else
47: #define DEBUGB(FLAG, CODE) { \
48:     if (debugflags::getflag (FLAG)) { \
49:         debugflags::where (FLAG, __FILE__, __LINE__, \
50:             __PRETTY_FUNCTION__); \
51:         cerr << CODE << endl; \
52:     } \
53: }
54: #define DEBUGF(FLAG, CODE) { \
55:     if (debugflags::getflag (FLAG)) { \
56:         debugflags::where (FLAG, __FILE__, __LINE__, \
57:             __PRETTY_FUNCTION__); \
58:         cerr << CODE << endl; \
59:     } \
60: }
61: #define DEBUGS(FLAG, STMT) { \
62:     if (debugflags::getflag (FLAG)) { \
63:         debugflags::where (FLAG, __FILE__, __LINE__, \
64:             __PRETTY_FUNCTION__); \
65:         STMT; \
66:     } \
67: }
68: #endif
69:
70: #endif
71:
```



```
1: // $Id: debug.cpp,v 1.4 2021-05-22 02:29:26-07 - - $
2: //Kai O'Brien (kimobrie@ucsc.edu)
3:
4: #include <climits>
5: #include <iostream>
6: using namespace std;
7:
8: #include "debug.h"
9: #include "util.h"
10:
11: debugflags::flagset debugflags::flags {};
12:
13: void debugflags::setflags (const string& initflags) {
14:     for (const unsigned char flag: initflags) {
15:         if (flag == '@') flags.set();
16:         else flags.set (flag, true);
17:     }
18: }
19:
20: // getflag -
21: //     Check to see if a certain flag is on.
22:
23: bool debugflags::getflag (char flag) {
24:     // WARNING: Don't TRACE this function or the stack will blow up.
25:     return flags.test (static_cast<unsigned char> (flag));
26: }
27:
28: void debugflags::where (char flag, const char* file, int line,
29:                        const char* pretty_function) {
30:     cout << sys_info::execname() << ": DEBUG(" << flag << ") "
31:          << file << "[" << line << "]" " << endl
32:          << "    " << pretty_function << endl;
33: }
34:
```

```
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 1.24 2021-05-23 00:08:49-07 - - $
2: //Kai O'Brien (kimobrie@ucsc.edu)
3:
4: #include <cstdlib>
5: #include <exception>
6: #include <iostream>
7: #include <string>
8: #include <unistd.h>
9: //-----
10: #include <cassert>
11: #include <cerrno>
12: #include <fstream>
13: #include <iomanip>
14: #include <regex>
15: #include <stdexcept>
16: #include <typeinfo>
17:
18: using namespace std;
19:
20: #include "listmap.h"
21: #include "xpair.h"
22: #include "util.h"
23:
24: using str_str_map = listmap<string, string>;
25: using str_str_pair = str_str_map::value_type;
26: str_str_map test; //listmap
27:
28: void scan_options (int argc, char** argv) {
29:     opterr = 0;
30:     for (;;) {
31:         int option = getopt (argc, argv, "@:");
32:         if (option == EOF) break;
33:         switch (option) {
34:             case '@':
35:                 debugflags::setflags (optarg);
36:                 break;
37:             default:
38:                 complain() << "-" << char (optopt) << ": invalid option"
39:                     << endl;
40:                 break;
41:         }
42:     }
43: }
44:
45: void whitespace(string *line){
46:     //trim leading whitespace and returns position of =sign or -1
47:     unsigned long first = 0;
48:     while(first<line->size() && line->at(first) == ' '){
49:         line->erase(first,1); //at first position
50:         if(line->at(first)=='='){
51:
52:         }
53:         ++first;
54:     }
55:     int mid = 1;
56:     while(first<line->size()){
57:         if(line->at(first)=='='){
58:         }
```

```
59:         if(line->at(first)=='\n'){
60:             line->erase(first,mid);//at first position
61:         }
62:         else{
63:             ++mid;
64:         }
65:         ++first;
66:     }
67:     //trims trailing whitespace
68:     ssize_t last = line->size()-1;
69:     while(last>0 &&line->at(last) == ' '){
70:         if(line->at(last)=='='){
71:             }
72:
73:         line->erase(last,line->size()-1);//at first position
74:         --last;
75:     }
76: }
77:
78: size_t eq_pos(string *line){
79:     size_t eq = 1234;
80:     size_t first = 0;
81:     while(first<line->size()){
82:         if(line->at(first) == '='){
83:             eq = first;
84:             break;
85:         }
86:         ++first;
87:     }
88:     return eq;
89: }
90:
91: //insert stuff to map when key = value not found
92: void catfile_helper (istream& infile, const string& filename) {
93:     static string colons (32, ':');
94:     // cout << colons << endl << filename << endl << colons << endl;
95:     regex comment_regex {R"(\s*(#.*)?$)"};
96:     regex key_value_regex {R"(\s*(.*?)\s*=\s*(.*?)\s*$)"};
97:     regex trimmed_regex {R"(\s*([^\s]+?)\s*$)"};
98:     int i = 1;
99:     for(;;) {
100:         string line;
101:         getline (infile, line);
102:         whitespace(&line);//trim whitespace
103:
104:         //cout << "input: \"" << line << "\"" << endl;
105:         if(line.length()>0){
106:             cout<< line;
107:             smatch result;
108:             if (regex_search (line, result, comment_regex)) {
109:                 cout<<filename<<": " <<i<<": " <<line<<endl;
110:
111:             }
112:
113:             else if (regex_search (line, result, key_value_regex)) {
114:                 cout<<filename<<": " <<i<<": " <<line<<endl;
115:                 // =
116:                 if(line.at(0)=='='){
```

```
117:         //if the = is the only thing
118:         if(line.size()==1){
119:             for (auto itor = test.begin(); itor != test.end(); ++i
tor) {
120:                 cout<< itor->first<< " = " <<itor->second<<endl;
121:             }
122:         }
123:         else{ //value
124:             for (auto itor = test.begin(); itor != test.end(); ++itor
) {
125:                 if(itor->second == result[2]){
126:                     cout<< itor->first<< " = " <<itor->second<<endl;
127:                 }
128:             }
129:         }
130:     }
131:     else if(line.at(line.size()-1)==' '){ //key =
132:         if(test.find(result[1])!=test.end() ){
133:             test.erase(test.find(result[1])); //valgrind
134:         }
135:         else{
136:             cout<< result[1]<< ": " <<"key not found"<<endl;
137:         }
138:     }
139:     else{//key = value
140:         test.insert(str_str_pair(result[1],result[2]));
141:     }
142: }
143: // key = , =, or =value
144: else if (regex_search (line, result, trimmed_regex)) {
145:     cout<<filename<<": " <<i<<": " <<line<<endl;
146:     size_t eq_pos1 = eq_pos(&line);
147:     //key
148:     if(eq_pos1==1234){
149:         auto it = test.find(result[1]);
150:         if(test.find(result[1])!=test.end()){
151:             cout<< it->first<< " = " <<it->second<<endl;
152:         }
153:         else{
154:             cout<< result[1]<< ": " <<"key not found"<<endl;
155:         }
156:     }
157: }else {
158:     assert (false and "This can not happen.");
159: }
160: i++;
161: }
162: if (infile.eof()) break;
163: }
164: }
165:
166: int main (int argc, char** argv) {
167:     sys_info::execname (argv[0]);
168:     scan_options (argc, argv);
169:     //-----matchlines
170:     const string cin_name = "-";
171:     int status = 0;
172:     string progname ( (argv[0]));
```



```
173:     vector<string> filenames (&argv[1], &argv[argc]);
174:     if (filenames.size() == 0) filenames.push_back (cin_name);
175:     for (const auto& filename: filenames) {
176:         if (filename == cin_name) catfile_helper (cin, filename);
177:         else {
178:             ifstream infile (filename);
179:             if (infile.fail()) {
180:                 status = 1;
181:                 cerr << progname << ": " << filename << ": "
182:                     << strerror (errno) << endl;
183:             }else {
184:                 catfile_helper (infile, filename);
185:                 infile.close();
186:             }
187:         }
188:     }
189:     return status;
190:     // cout << "EXIT_SUCCESS" << endl;
191:     // return EXIT_SUCCESS;
192: }
193:
```

```
1: # $Id: Makefile,v 1.27 2021-05-22 02:29:26-07 - - $
2: #Kai O'Brien (kimobrie@ucsc.edu)
3:
4: MKFILE      = Makefile
5: DEFILE      = ${MKFILE}.dep
6: NOINCL      = ci clean spotless check lint
7: NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
8: GMAKE       = ${MAKE} --no-print-directory
9:
10: GPPWARN      = -Wall -Wextra -Wpedantic -Wshadow -Wold-style-cast
11: GPPOPTS      = ${GPPWARN} -fdiagnostics-color=never
12: COMPILECPP   = g++ -std=gnu++17 -g -O0 ${GPPOPTS}
13: MAKEDEPCPP   = g++ -std=gnu++17 -MM ${GPPOPTS}
14: UTILBIN      = /afs/cats.ucsc.edu/courses/cse111-wm/bin
15:
16: MODULES      = listmap xless xpair debug util main
17: CPPSOURCE    = ${wildcard ${MODULES:=.cpp}}
18: OBJECTS      = ${CPPSOURCE:.cpp=.o}
19: SOURCELIST   = ${foreach MOD, ${MODULES}, ${MOD}.h ${MOD}.tcc ${MOD}.cpp}
20: ALLSOURCE    = ${wildcard ${SOURCELIST}}
21: EXECBIN      = keyvalue
22: OTHERS       = ${MKFILE} ${DEFILE}
23: ALLSOURCES   = ${ALLSOURCE} ${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: lint : ${CPPSOURCE}
35:             ${UTILBIN}/cpplint.py.perl ${CPPSOURCE}
36:
37: check : ${ALLSOURCES}
38:             ${UTILBIN}/checksource ${ALLSOURCES}
39:
40: ci : ${ALLSOURCES}
41:             ${UTILBIN}/cid -is ${ALLSOURCES}
42:
43: lis : ${ALLSOURCES}
44:             mkpspdf ${LISTING} ${ALLSOURCES}
45:
46: clean :
47:         - rm ${OBJECTS} ${DEFILE} core
48:
49: spotless : clean
50:         - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
51:
52: dep : ${ALLCPPSRC}
53:         @ echo "# ${DEFILE} created `LC_TIME=C date`" >${DEFILE}
54:         ${MAKEDEPCPP} ${CPPSOURCE} >>${DEFILE}
55:
56: ${DEFILE} :
57:         @ touch ${DEFILE}
58:         ${GMAKE} dep
```

```
59:
60: again :
61:     ${GMAKE} spotless dep ci all lis
62: submit:
63:     submit cse111-wm.s21 ko3 *.cpp *.tcc *.h Makefile README
64:
65: ifeq (${NEEDINCL}, )
66: include ${DEPFILE}
67: endif
68:
```

```
1: # Makefile.dep created Sun May 23 00:08:49 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
```