```
 1: // $Id: commands.h,v 1.5 2014-03-26 18:39:40-07 - - $
 2:
 3: #ifndef __COMMANDS_H__
 4: #define __COMMANDS_H__
 5:
 6: #include <map>
 7:
 8: using namespace std;
 9:
10: #include "inode.h"
11: #include "util.h"
12:
13: //
14: // A couple of convenient typedefs to avoid verbosity.
15: //
16:
17: typedef void (*function) (inode_state& state, const wordvec& words);
18: typedef map<string, function> commandmap;
19:
20: //
21: // commands -
22: //    A class to hold and dispatch each of the command functions.
23: //    Each command "foo" is interpreted by a function fn_foo.
24: // ctor -
25: //    The default ctor initializes the map.
26: // operator[] -
27: //    Given a string, returns a function associated with it,
28: //    or 0 if not found.
29: //
30:
31: class commands {
32:    private:
33:       commands (const inode&) = delete; // copy ctor
34:       commands& operator= (const inode&) = delete; // operator=
35:       commandmap map;
36:    public:
37:       commands();
38:       function at (const string& cmd);
39: };
40:
```

```
41:
42: //
43: // execution functions -
44: //    See the man page for a description of each of these functions.
45: //
46:
47: void fn_cat    (inode_state& state, const wordvec& words);
48: void fn_cd     (inode_state& state, const wordvec& words);
49: void fn_echo   (inode_state& state, const wordvec& words);
50: void fn_exit   (inode_state& state, const wordvec& words);
51: void fn_ls     (inode_state& state, const wordvec& words);
52: void fn_lsr    (inode_state& state, const wordvec& words);
53: void fn_make   (inode_state& state, const wordvec& words);
54: void fn_mkdir  (inode_state& state, const wordvec& words);
55: void fn_prompt (inode_state& state, const wordvec& words);
56: void fn_pwd    (inode_state& state, const wordvec& words);
57: void fn_rm     (inode_state& state, const wordvec& words);
58: void fn_rmr    (inode_state& state, const wordvec& words);
59:
60: //
61: // exit_status_message -
62: //    Prints an exit message and returns the exit status, as recorded
63: //    by any of the functions.
64: //
65:
66: int exit_status_message();
67: class ysh_exit_exn: public exception {};
68:
69: #endif
70:
```

```
 1: // $Id: debug.h,v 1.3 2014-03-26 19:55:18-07 - - $
 2:
 3: #ifndef __DEBUG_H__
 4: #define __DEBUG_H__
 5:
 6: #include <string>
 7: #include <vector>
 8:
 9: using namespace std;
10:
11: //
12: // debug -
13: //    static class for maintaining global debug flags, each indicated
14: //    by a single character.
15: // setflags -
16: //    Takes a string argument, and sets a flag for each char in the
17: //    string.  As a special case, '@', sets all flags.
18: // getflag -
19: //    Used by the DEBUGF macro to check to see if a flag has been set.
20: //    Not to be called by user code.
21: //
22:
23: class debugflags {
24:    private:
25:       static vector<bool> flags;
26:    public:
27:       static void setflags (const string& optflags);
28:       static bool getflag (char flag);
29:       static void where (char flag, const char* file, int line,
30:                          const char* func);
31: };
32:
```

```
33:
34: //
35: // DEBUGF -
36: //    Macro which expands into trace code.  First argument is a
37: //    trace flag char, second argument is output code that can
38: //    be sandwiched between <<.  Beware of operator precedence.
39: //    Example:
40: //       DEBUGF ('u', "foo = " << foo);
41: //    will print two words and a newline if flag 'u' is  on.
42: //    Traces are preceded by filename, line number, and function.
43: //
44:
45: #ifdef NDEBUG
46: #define DEBUGF(FLAG,CODE) ;
47: #define DEBUGS(FLAG,STMT) ;
48: #else
49: #define DEBUGF(FLAG,CODE) { \
50:            if (debugflags::getflag (FLAG)) { \
51:               debugflags::where (FLAG, __FILE__, __LINE__, __func__); \
52:               cerr << CODE << endl; \
53:            } \
54:         }
55: #define DEBUGS(FLAG,STMT) { \
56:            if (debugflags::getflag (FLAG)) { \
57:               debugflags::where (FLAG, __FILE__, __LINE__, __func__); \
58:               STMT; \
59:            } \
60:         }
61: #endif
62:
63: #endif
64:
```

```
 1: // $Id: inode.h,v 1.6 2014-03-26 19:55:18-07 - - $
 2:
 3: #ifndef __INODE_H__
 4: #define __INODE_H__
 5:
 6: #include <exception>
 7: #include <iostream>
 8: #include <map>
 9: #include <vector>
10:
11: using namespace std;
12:
13: #include "util.h"
14:
15: //
16: // inode_t -
17: //    An inode is either a directory or a plain file.
18: //
19:
20: enum inode_t {DIR_INODE, FILE_INODE};
21:
22: //
23: // directory -
24: //    A directory is a list of paired strings (filenames) and inodes.
25: //    An inode in a directory may be a directory or a file.
26: //
27:
28: class inode;
29: typedef map<string, inode*> directory;
30:
31: //
32: // inode_state -
33: //    A small convenient class to maintain the state of the simulated
34: //    process:  the root (/), the current directory (.), and the
35: //    prompt.
36: //
37:
38: class inode_state {
39:    friend class inode;
40:    friend ostream& operator<< (ostream& out, const inode_state&);
41:    private:
42:       inode_state (const inode_state&) = delete; // copy ctor
43:       inode_state& operator= (const inode_state&) = delete; // op=
44:       inode* root;
45:       inode* cwd;
46:       string prompt;
47:    public:
48:       inode_state();
49: };
50:
51: ostream& operator<< (ostream& out, const inode_state&);
52:
```

```
53:
54: //
55: // class inode -
56: //
57: // inode ctor -
58: //    Create a new inode of the given type, using a union.
59: // get_inode_nr -
60: //    Retrieves the serial number of the inode.  Inode numbers are
61: //    allocated in sequence by small integer.
62: // size -
63: //    Returns the size of an inode.  For a directory, this is the
64: //    number of dirents.  For a text file, the number of characters
65: //    when printed (the sum of the lengths of each word, plus the
66: //    number of words.
67: // readfile -
68: //    Returns a copy of the contents of the wordvec in the file.
69: //    Throws an yshell_exn for a directory.
70: // writefile -
71: //    Replaces the contents of a file with new contents.
72: //    Throws an yshell_exn for a directory.
73: // remove -
74: //    Removes the file or subdirectory from the current inode.
75: //    Throws an yshell_exn if this is not a directory, the file
76: //    does not exist, or the subdirectory is not empty.
77: //    Here empty means the only entries are dot (.) and dotdot (..).
78: // mkdir -
79: //    Creates a new directory under the current directory and
80: //    immediately adds the directories dot (.) and dotdot (..) to it.
81: //    Note that the parent (..) of / is / itself.  It is an error
82: //    if the entry already exists.
83: // mkfile -
84: //    Create a new empty text file with the given name.  Error if
85: //    a dirent with that name exists.
86: //
87:
```

```
 88:
 89: class inode {
 90:    friend class inode_state;
 91:    private:
 92:       int inode_nr;
 93:       inode_t type;
 94:       union {
 95:          directory* dirents;
 96:          wordvec* data;
 97:       } contents;
 98:       static int next_inode_nr;
 99:    public:
100:       inode (inode_t init_type);
101:       inode (const inode& source);
102:       inode& operator= (const inode& from);
103:       int get_inode_nr() const;
104:       int size() const;
105:       const wordvec& readfile() const;
106:       void writefile (const wordvec& newdata);
107:       void remove (const string& filename);
108:       inode& mkdir (const string& dirname);
109:       inode& mkfile (const string& filename);
110: };
111:
112: #endif
113:
```

```
 1: // $Id: util.h,v 1.6 2014-03-26 19:55:18-07 - - $
 2:
 3: //
 4: // util -
 5: //    A utility class to provide various services not conveniently
 6: //    included in other modules.
 7: //
 8:
 9: #ifndef __UTIL_H__
10: #define __UTIL_H__
11:
12: #include <iostream>
13: #include <string>
14: #include <vector>
15:
16: #ifdef __GNUC__
17: #include <stdexcept>
18: #endif
19:
20: using namespace std;
21:
22: //
23: // Convenient typedef to allow brevity of code elsewhere.
24: //
25:
26: typedef vector<string> wordvec;
27:
28: //
29: // yshell_exn -
30: //    Extend runtime_error for throwing exceptions related to this
31: //    program.
32: //
33:
34: class yshell_exn: public runtime_error {
35:    public:
36:       explicit yshell_exn (const string& what);
37: };
38:
39: //
40: // setexecname -
41: //    Sets the static string to be used as an execname.
42: // execname -
43: //    Returns the basename of the executable image, which is used in
44: //    printing error messags.
45: //
46:
47: void setexecname (const string&);
48: string& execname();
49:
```

```
 50:
 51: //
 52: // want_echo -
 53: //    We want to echo all of cin to cout if either cin or cout
 54: //    is not a tty.  This helps make batch processing easier by
 55: //    making cout look like a terminal session trace.
 56: //
 57:
 58: bool want_echo();
 59:
 60: //
 61: // exit_status -
 62: //    A static class for maintaining the exit status.  The default
 63: //    status is EXIT_SUCCESS (0), but can be set to another value,
 64: //    such as EXIT_FAILURE (1) to indicate that error messages have
 65: //    been printed.
 66: //
 67:
 68: class exit_status {
 69:    private:
 70:       static int status;
 71:    public:
 72:       static void set (int);
 73:       static int get();
 74: };
 75:
 76: //
 77: // split -
 78: //    Split a string into a wordvec (as defined above).  Any sequence
 79: //    of chars in the delimiter string is used as a separator.  To
 80: //    Split a pathname, use "/".  To split a shell command, use " ".
 81: //
 82:
 83: wordvec split (const string& line, const string& delimiter);
 84:
 85: // complain -
 86: //    Used for starting error messages.  Sets the exit status to
 87: //    EXIT_FAILURE, writes the program name to cerr, and then
 88: //    returns the cerr ostream.  Example:
 89: //       complain() << filename << ": some problem" << endl;
 90: //
 91:
 92: ostream& complain();
 93:
 94: //
 95: // operator<< (vector) -
 96: //    An overloaded template operator which allows vectors to be
 97: //    printed out as a single operator, each element separated from
 98: //    the next with spaces.  The item_t must have an output operator
 99: //    defined for it.
100: //
101:
102: template <typename item_t>
103: ostream& operator<< (ostream& out, const vector<item_t>& vec);
104:
105: #include "util.tcc"
106: #endif
107:
```

```
 1: // $Id: util.tcc,v 1.1 2014-03-26 17:34:27-07 - - $
 2:
 3: template <typename item_t>
 4: ostream& operator<< (ostream& out, const vector<item_t>& vec) {
 5:    bool want_space = false;
 6:    for (typename vector<item_t>::const_iterator itor = vec.cbegin();
 7:         itor != vec.cend(); ++itor) {
 8:       if (want_space) out << " ";
 9:                  else want_space = true;
10:       out << *itor;
11:    }
12:    return out;
13: }
14:
```

```
   1: // $Id: commands.cpp,v 1.9 2014-03-26 19:55:18-07 - - $
   2:
   3: #include "commands.h"
   4: #include "debug.h"
   5:
   6: commands::commands(): map ({
   7:    {"cat"   , fn_cat   },
   8:    {"cd"    , fn_cd    },
   9:    {"echo"  , fn_echo  },
  10:    {"exit"  , fn_exit  },
  11:    {"ls"    , fn_ls    },
  12:    {"lsr"   , fn_lsr   },
  13:    {"make"  , fn_make  },
  14:    {"mkdir" , fn_mkdir },
  15:    {"prompt", fn_prompt},
  16:    {"pwd"   , fn_pwd   },
  17:    {"rm"    , fn_rm    },
  18: }){}
  19:
  20: function commands::at (const string& cmd) {
  21:    // Note: value_type is pair<const key_type, mapped_type>
  22:    // So: iterator->first is key_type (string)
  23:    // So: iterator->second is mapped_type (function)
  24:    commandmap::const_iterator result = map.find (cmd);
  25:    if (result == map.end()) {
  26:       throw yshell_exn (cmd + ": no such function");
  27:    }
  28:    return result->second;
  29: }
  30:
```

```
31:
32: void fn_cat (inode_state& state, const wordvec& words){
33:    DEBUGF ('c', state);
34:    DEBUGF ('c', words);
35: }
36:
37: void fn_cd (inode_state& state, const wordvec& words){
38:    DEBUGF ('c', state);
39:    DEBUGF ('c', words);
40: }
41:
42: void fn_echo (inode_state& state, const wordvec& words){
43:    DEBUGF ('c', state);
44:    DEBUGF ('c', words);
45: }
46:
47: void fn_exit (inode_state& state, const wordvec& words){
48:    DEBUGF ('c', state);
49:    DEBUGF ('c', words);
50:    throw ysh_exit_exn ();
51: }
52:
53: void fn_ls (inode_state& state, const wordvec& words){
54:    DEBUGF ('c', state);
55:    DEBUGF ('c', words);
56: }
57:
58: void fn_lsr (inode_state& state, const wordvec& words){
59:    DEBUGF ('c', state);
60:    DEBUGF ('c', words);
61: }
62:
```

```
63:
64: void fn_make (inode_state& state, const wordvec& words){
65:    DEBUGF ('c', state);
66:    DEBUGF ('c', words);
67: }
68:
69: void fn_mkdir (inode_state& state, const wordvec& words){
70:    DEBUGF ('c', state);
71:    DEBUGF ('c', words);
72: }
73:
74: void fn_prompt (inode_state& state, const wordvec& words){
75:    DEBUGF ('c', state);
76:    DEBUGF ('c', words);
77: }
78:
79: void fn_pwd (inode_state& state, const wordvec& words){
80:    DEBUGF ('c', state);
81:    DEBUGF ('c', words);
82: }
83:
84: void fn_rm (inode_state& state, const wordvec& words){
85:    DEBUGF ('c', state);
86:    DEBUGF ('c', words);
87: }
88:
89: void fn_rmr (inode_state& state, const wordvec& words){
90:    DEBUGF ('c', state);
91:    DEBUGF ('c', words);
92: }
93:
94: int exit_status_message() {
95:    int exit_status = exit_status::get();
96:    cout << execname() << ": exit(" << exit_status << ")" << endl;
97:    return exit_status;
98: }
99:
```

```
 1: // $Id: debug.cpp,v 1.4 2014-03-26 19:49:30-07 - - $
 2:
 3: #include <cassert>
 4: #include <climits>
 5: #include <iostream>
 6: #include <vector>
 7:
 8: using namespace std;
 9:
10: #include "debug.h"
11: #include "util.h"
12:
13: vector<bool> debugflags::flags (UCHAR_MAX + 1, false);
14:
15: void debugflags::setflags (const string& initflags) {
16:    for (const char flag: initflags) {
17:       if (flag == '@') flags.assign (flags.size(), true);
18:                   else flags[flag] = true;
19:    }
20:    // Note that DEBUGF can trace setflags.
21:    if (getflag ('x')) {
22:       string flag_chars;
23:       for (size_t index = 0; index < flags.size(); ++index) {
24:          if (getflag (index)) flag_chars += (char) index;
25:       }
26:       DEBUGF ('x', "debugflags::flags = " << flag_chars);
27:    }
28: }
29:
30: //
31: // getflag -
32: //    Check to see if a certain flag is on.
33: //
34:
35: bool debugflags::getflag (char flag) {
36:    // WARNING: Don't TRACE this function or the stack will blow up.
37:    unsigned uflag = (unsigned char) flag;
38:    assert (uflag < flags.size());
39:    return flags[uflag];
40: }
41:
42: void debugflags::where (char flag, const char* file, int line,
43:                         const char* func) {
44:    cout << execname() << ": DEBUG(" << flag << ") "
45:         << file << "[" << line << "] " << func << "()" << endl;
46: }
47:
```

```
 1: // $Id: inode.cpp,v 1.3 2014-03-26 18:39:40-07 - - $
 2:
 3: #include <cassert>
 4: #include <iostream>
 5:
 6: using namespace std;
 7:
 8: #include "debug.h"
 9: #include "inode.h"
10:
11: int inode::next_inode_nr = 1;
12:
13: inode::inode(inode_t init_type):
14:    inode_nr (next_inode_nr++), type (init_type)
15: {
16:    switch (type) {
17:       case DIR_INODE:
18:            contents.dirents = new directory();
19:            break;
20:       case FILE_INODE:
21:            contents.data = new wordvec();
22:            break;
23:    }
24:    DEBUGF ('i', "inode " << inode_nr << ", type = " << type);
25: }
26:
27: //
28: // copy ctor -
29: //    Make a copy of a given inode.  This should not be used in
30: //    your program if you can avoid it, since it is expensive.
31: //    Here, we can leverage operator=.
32: //
33: inode::inode (const inode& that) {
34:    *this = that;
35: }
36:
37: //
38: // operator= -
39: //    Assignment operator.  Copy an inode.  Make a copy of a
40: //    given inode.  This should not be used in your program if
41: //    you can avoid it, since it is expensive.
42: //
43: inode& inode::operator= (const inode& that) {
44:    if (this != &that) {
45:       inode_nr = that.inode_nr;
46:       type = that.type;
47:       contents = that.contents;
48:    }
49:    DEBUGF ('i', "inode " << inode_nr << ", type = " << type);
50:    return *this;
51: }
52:
```

```
53:
54: int inode::get_inode_nr() const {
55:    DEBUGF ('i', "inode = " << inode_nr);
56:    return inode_nr;
57: }
58:
59: int inode::size() const {
60:    int size = 0;
61:    DEBUGF ('i', "size = " << size);
62:    return size;
63: }
64:
65: const wordvec& inode::readfile() const {
66:    DEBUGF ('i', *contents.data);
67:    assert (type == FILE_INODE);
68:    return *contents.data;
69: }
70:
71: void inode::writefile (const wordvec& words) {
72:    DEBUGF ('i', words);
73:    assert (type == FILE_INODE);
74: }
75:
76: void inode::remove (const string& filename) {
77:    DEBUGF ('i', filename);
78:    assert (type == DIR_INODE);
79: }
80:
81: inode_state::inode_state(): root (NULL), cwd (NULL), prompt ("%") {
82:    DEBUGF ('i', "root = " << (void*) root << ", cwd = " << (void*) cwd
83:           << ", prompt = " << prompt);
84: }
85:
86: ostream& operator<< (ostream& out, const inode_state& state) {
87:    out << "inode_state: root = " << state.root
88:        << ", cwd = " << state.cwd;
89:    return out;
90: }
91:
```

```
 1: // $Id: util.cpp,v 1.8 2014-03-26 18:39:40-07 - - $
 2:
 3: #include <cstdlib>
 4: #include <unistd.h>
 5:
 6: using namespace std;
 7:
 8: #include "util.h"
 9: #include "debug.h"
10:
11: yshell_exn::yshell_exn (const string& what): runtime_error (what) {
12: }
13:
14: int exit_status::status = EXIT_SUCCESS;
15: static string execname_string;
16:
17: void exit_status::set (int new_status) {
18:    status = new_status;
19: }
20:
21: int exit_status::get() {
22:    return status;
23: }
24:
25: void setexecname (const string& name) {
26:    execname_string =  name.substr (name.rfind ('/') + 1);
27:    DEBUGF ('u', execname_string);
28: }
29:
30: string& execname() {
31:    return execname_string;
32: }
33:
34: bool want_echo() {
35:    const int CIN_FD = 0;
36:    const int COUT_FD = 1;
37:    bool cin_isatty = isatty (CIN_FD);
38:    bool cout_isatty = isatty (COUT_FD);
39:    DEBUGF ('u', "cin_isatty = " << cin_isatty
40:           << ", cout_isatty = " << cout_isatty);
41:    return ! cin_isatty || ! cout_isatty;
42: }
43:
```

```
44:
45: wordvec split (const string& line, const string& delimiters) {
46:    wordvec words;
47:    size_t end = 0;
48:
49:    // Loop over the string, splitting out words, and for each word
50:    // thus found, append it to the output wordvec.
51:    for (;;) {
52:       size_t start = line.find_first_not_of (delimiters, end);
53:       if (start == string::npos) break;
54:       end = line.find_first_of (delimiters, start);
55:       words.push_back (line.substr (start, end - start));
56:    }
57:    DEBUGF ('u', words);
58:    return words;
59: }
60:
61: ostream& complain() {
62:    exit_status::set (EXIT_FAILURE);
63:    cerr << execname() << ": ";
64:    return cerr;
65: }
66:
```

```
 1: // $Id: main.cpp,v 1.1 2014-03-26 19:51:59-07 - - $
 2:
 3: #include <cstdlib>
 4: #include <iostream>
 5: #include <string>
 6: #include <utility>
 7: #include <unistd.h>
 8:
 9: using namespace std;
10:
11: #include "commands.h"
12: #include "debug.h"
13: #include "inode.h"
14: #include "util.h"
15:
16: //
17: // scan_options
18: //    Options analysis:  The only option is -Dflags.
19: //
20:
21: void scan_options (int argc, char** argv) {
22:    opterr = 0;
23:    for (;;) {
24:       int option = getopt (argc, argv, "@:");
25:       if (option == EOF) break;
26:       switch (option) {
27:          case '@':
28:             debugflags::setflags (optarg);
29:             break;
30:          default:
31:             complain() << "-" << (char) option << ": invalid option"
32:                        << endl;
33:             break;
34:       }
35:    }
36:    if (optind < argc) {
37:       complain() << "operands not permitted" << endl;
38:    }
39: }
40:
```

```
41:
42: //
43: // main -
44: //    Main program which loops reading commands until end of file.
45: //
46:
47: int main (int argc, char** argv) {
48:    setexecname (argv[0]);
49:    cout << boolalpha; // Print false or true instead of 0 or 1.
50:    cerr << boolalpha;
51:    cout << argv[0] << " build " << __DATE__ << " " << __TIME__ << endl;
52:    scan_options (argc, argv);
53:    bool need_echo = want_echo();
54:    commands cmdmap;
55:    string prompt = "%";
56:    inode_state state;
57:    try {
58:       for (;;) {
59:          try {
60:
61:             // Read a line, break at EOF, and echo print the prompt
62:             // if one is needed.
63:             cout << prompt << " ";
64:             string line;
65:             getline (cin, line);
66:             if (cin.eof()) {
67:                if (need_echo) cout << "^D";
68:                cout << endl;
69:                DEBUGF ('y', "EOF");
70:                break;
71:             }
72:             if (need_echo) cout << line << endl;
73:
74:             // Split the line into words and lookup the appropriate
75:             // function.  Complain or call it.
76:             wordvec words = split (line, " \t");
77:             DEBUGF ('y', "words = " << words);
78:             function fn = cmdmap.at(words.at(0));
79:             fn (state, words);
80:          }catch (yshell_exn& exn) {
81:             // If there is a problem discovered in any function, an
82:             // exn is thrown and printed here.
83:             complain() << exn.what() << endl;
84:          }
85:       }
86:    } catch (ysh_exit_exn& ) {
87:       // This catch intentionally left blank.
88:    }
89:
90:    return exit_status_message();
91: }
92:
```

```
 1: # $Id: Makefile,v 1.8 2014-03-26 19:55:18-07 - - $
 2:
 3: MKFILE      = Makefile
 4: DEPFILE     = ${MKFILE}.deps
 5: NOINCL      = ci clean spotless
 6: NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
 7: GMAKE       = ${MAKE} --no-print-directory
 8:
 9: COMPILECPP  = g++ -g -O0 -Wall -Wextra -std=gnu++0x
10: MAKEDEPCPP  = g++ -MM
11:
12: CPPSOURCE   = commands.cpp debug.cpp inode.cpp util.cpp main.cpp
13: CPPHEADER   = commands.h debug.h inode.h util.h util.tcc
14: EXECBIN     = yshell
15: OBJECTS     = ${CPPSOURCE:.cpp=.o}
16: OTHERS      = ${MKFILE} README
17: ALLSOURCES  = ${CPPHEADER} ${CPPSOURCE} ${OTHERS}
18: LISTING     = Listing.code.ps
19: CLASS       = cmps109-wm.s14
20: PROJECT     = asg1
21:
22: all : ${EXECBIN}
23:         - checksource ${ALLSOURCES}
24:
25: ${EXECBIN} : ${OBJECTS}
26:         ${COMPILECPP} -o $@ ${OBJECTS}
27:
28: %.o : %.cpp
29:         ${COMPILECPP} -c $<
30:
31: ci : ${ALLSOURCES}
32:         cid + ${ALLSOURCES}
33:         - checksource ${ALLSOURCES}
34:
35: lis : ${ALLSOURCES}
36:         mkpspdf ${LISTING} ${ALLSOURCES} ${DEPFILE}
37:
38: clean :
39:         - rm ${OBJECTS} ${DEPFILE} core ${EXECBIN}.errs
40:
41: spotless : clean
42:         - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
43:
```

```
44:
45: submit : ${ALLSOURCES}
46:          - checksource ${ALLSOURCES}
47:          submit ${CLASS} ${PROJECT} ${ALLSOURCES}
48:
49: deps : ${CPPSOURCE} ${CPPHEADER}
50:          @ echo "# ${DEPFILE} created `LC_TIME=C date`" >${DEPFILE}
51:          ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPFILE}
52:
53: ${DEPFILE} :
54:          @ touch ${DEPFILE}
55:          ${GMAKE} deps
56:
57: again :
58:          ${GMAKE} spotless deps ci all lis
59:
60: ifeq (${NEEDINCL}, )
61: include ${DEPFILE}
62: endif
63:
```

```
1: $Id: README,v 1.1 2013-06-18 17:32:08-07 - - $
```

```
1: # Makefile.deps created Thu Mar 27 16:26:30 PDT 2014
2: commands.o: commands.cpp commands.h inode.h util.h util.tcc debug.h
3: debug.o: debug.cpp debug.h util.h util.tcc
4: inode.o: inode.cpp debug.h inode.h util.h util.tcc
5: util.o: util.cpp util.h util.tcc debug.h
6: main.o: main.cpp commands.h inode.h util.h util.tcc debug.h
```