```
 1: // $Id: commands.h,v 1.8 2014-06-11 13:49:31-07 - - $
 2:
 3: #ifndef __COMMANDS_H__
 4: #define __COMMANDS_H__
 5:
 6: #include <map>
 7: using namespace std;
 8:
 9: #include "inode.h"
10: #include "util.h"
11:
12: //
13: // A couple of convenient usings to avoid verbosity.
14: //
15:
16: using command_fn = void (*)(inode_state& state, const wordvec& words);
17: using command_map = map<string,command_fn>;
18:
19: //
20: // commands -
21: //    A class to hold and dispatch each of the command functions.
22: //    Each command "foo" is interpreted by a command_fn fn_foo.
23: // ctor -
24: //    The default ctor initializes the map.
25: // operator[] -
26: //    Given a string, returns a command_fn associated with it,
27: //    or 0 if not found.
28: //
29:
30: class commands {
31:    private:
32:       commands (const inode&) = delete; // copy ctor
33:       commands& operator= (const inode&) = delete; // operator=
34:       command_map map;
35:    public:
36:       commands();
37:       command_fn at (const string& cmd);
38: };
39:
```

```
40:
41: //
42: // execution functions -
43: //     See the man page for a description of each of these functions.
44: //
45:
46: void fn_cat    (inode_state& state, const wordvec& words);
47: void fn_cd     (inode_state& state, const wordvec& words);
48: void fn_echo   (inode_state& state, const wordvec& words);
49: void fn_exit   (inode_state& state, const wordvec& words);
50: void fn_ls     (inode_state& state, const wordvec& words);
51: void fn_lsr    (inode_state& state, const wordvec& words);
52: void fn_make   (inode_state& state, const wordvec& words);
53: void fn_mkdir  (inode_state& state, const wordvec& words);
54: void fn_prompt (inode_state& state, const wordvec& words);
55: void fn_pwd    (inode_state& state, const wordvec& words);
56: void fn_rm     (inode_state& state, const wordvec& words);
57: void fn_rmr    (inode_state& state, const wordvec& words);
58:
59: //
60: // exit_status_message -
61: //     Prints an exit message and returns the exit status, as recorded
62: //     by any of the functions.
63: //
64:
65: int exit_status_message();
66: class ysh_exit_exn: public exception {};
67:
68: #endif
69:
```

```
 1: // $Id: debug.h,v 1.6 2015-01-02 18:13:23-08 - - $
 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: //
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:       using flagset = bitset<UCHAR_MAX + 1>;
26:       static flagset flags;
27:    public:
28:       static void setflags (const string& optflags);
29:       static bool getflag (char flag);
30:       static void where (char flag, const char* file, int line,
31:                          const char* func);
32: };
33:
```

```
34:
35: //
36: // DEBUGF -
37: //      Macro which expands into trace code.  First argument is a
38: //      trace flag char, second argument is output code that can
39: //      be sandwiched between <<.  Beware of operator precedence.
40: //      Example:
41: //         DEBUGF ('u', "foo = " << foo);
42: //      will print two words and a newline if flag 'u' is  on.
43: //      Traces are preceded by filename, line number, and function.
44: //
45:
46: #ifdef NDEBUG
47: #define DEBUGF(FLAG,CODE) ;
48: #define DEBUGS(FLAG,STMT) ;
49: #else
50: #define DEBUGF(FLAG,CODE) { \
51:           if (debugflags::getflag (FLAG)) { \
52:              debugflags::where (FLAG, __FILE__, __LINE__, __func__); \
53:              cerr << CODE << endl; \
54:           } \
55:        }
56: #define DEBUGS(FLAG,STMT) { \
57:           if (debugflags::getflag (FLAG)) { \
58:              debugflags::where (FLAG, __FILE__, __LINE__, __func__); \
59:              STMT; \
60:           } \
61:        }
62: #endif
63:
64: #endif
65:
```

```
 1: // $Id: inode.h,v 1.13 2014-06-12 18:10:25-07 - - $
 2:
 3: #ifndef __INODE_H__
 4: #define __INODE_H__
 5:
 6: #include <exception>
 7: #include <iostream>
 8: #include <memory>
 9: #include <map>
10: #include <vector>
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 {PLAIN_INODE, DIR_INODE};
21: class inode;
22: class file_base;
23: class plain_file;
24: class directory;
25: using inode_ptr = shared_ptr<inode>;
26: using file_base_ptr = shared_ptr<file_base>;
27: using plain_file_ptr = shared_ptr<plain_file>;
28: using directory_ptr = shared_ptr<directory>;
29:
30: //
31: // inode_state -
32: //    A small convenient class to maintain the state of the simulated
33: //    process:  the root (/), the current directory (.), and the
34: //    prompt.
35: //
36:
37: class inode_state {
38:    friend class inode;
39:    friend ostream& operator<< (ostream& out, const inode_state&);
40:    private:
41:       inode_state (const inode_state&) = delete; // copy ctor
42:       inode_state& operator= (const inode_state&) = delete; // op=
43:       inode_ptr root {nullptr};
44:       inode_ptr cwd {nullptr};
45:       string prompt {"% "};
46:    public:
47:       inode_state();
48: };
49:
```

```
 50:
 51: //
 52: // class inode -
 53: //
 54: // inode ctor -
 55: //     Create a new inode of the given type.
 56: // get_inode_nr -
 57: //     Retrieves the serial number of the inode.  Inode numbers are
 58: //     allocated in sequence by small integer.
 59: // size -
 60: //     Returns the size of an inode.  For a directory, this is the
 61: //     number of dirents.  For a text file, the number of characters
 62: //     when printed (the sum of the lengths of each word, plus the
 63: //     number of words.
 64: //
 65:
 66: class inode {
 67:    friend class inode_state;
 68:    private:
 69:       static int next_inode_nr;
 70:       int inode_nr;
 71:       inode_t type;
 72:       file_base_ptr contents;
 73:    public:
 74:       inode (inode_t init_type);
 75:       int get_inode_nr() const;
 76: };
 77:
 78: //
 79: // class file_base -
 80: //
 81: // Just a base class at which an inode can point.  No data or
 82: // functions.  Makes the synthesized members useable only from
 83: // the derived classes.
 84: //
 85:
 86: class file_base {
 87:    protected:
 88:       file_base () = default;
 89:       file_base (const file_base&) = default;
 90:       file_base (file_base&&) = default;
 91:       file_base& operator= (const file_base&) = default;
 92:       file_base& operator= (file_base&&) = default;
 93:       virtual ˜file_base () = default;
 94:       virtual size_t size() const = 0;
 95:    public:
 96:       friend plain_file_ptr plain_file_ptr_of (file_base_ptr);
 97:       friend directory_ptr directory_ptr_of (file_base_ptr);
 98: };
 99:
```

```
100:
101: //
102: // class plain_file -
103: //
104: // Used to hold data.
105: // synthesized default ctor -
106: //    Default vector<string> is a an empty vector.
107: // readfile -
108: //    Returns a copy of the contents of the wordvec in the file.
109: //    Throws an yshell_exn for a directory.
110: // writefile -
111: //    Replaces the contents of a file with new contents.
112: //    Throws an yshell_exn for a directory.
113: //
114:
115: class plain_file: public file_base {
116:    private:
117:       wordvec data;
118:    public:
119:       size_t size() const override;
120:       const wordvec& readfile() const;
121:       void writefile (const wordvec& newdata);
122: };
123:
124: //
125: // class directory -
126: //
127: // Used to map filenames onto inode pointers.
128: // default ctor -
129: //    Creates a new map with keys "." and "..".
130: // remove -
131: //    Removes the file or subdirectory from the current inode.
132: //    Throws an yshell_exn if this is not a directory, the file
133: //    does not exist, or the subdirectory is not empty.
134: //    Here empty means the only entries are dot (.) and dotdot (..).
135: // mkdir -
136: //    Creates a new directory under the current directory and
137: //    immediately adds the directories dot (.) and dotdot (..) to it.
138: //    Note that the parent (..) of / is / itself.  It is an error
139: //    if the entry already exists.
140: // mkfile -
141: //    Create a new empty text file with the given name.  Error if
142: //    a dirent with that name exists.
143:
144: class directory: public file_base {
145:    private:
146:       map<string,inode_ptr> dirents;
147:    public:
148:       size_t size() const override;
149:       void remove (const string& filename);
150:       inode& mkdir (const string& dirname);
151:       inode& mkfile (const string& filename);
152: };
153:
154: #endif
155:
```

```
 1: // $Id: util.h,v 1.9 2014-06-12 16:44:08-07 - - $
 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 <stdexcept>
14: #include <string>
15: #include <vector>
16: using namespace std;
17:
18: //
19: // Convenient type using to allow brevity of code elsewhere.
20: //
21:
22: using wordvec = vector<string>;
23:
24: //
25: // yshell_exn -
26: //     Extend runtime_error for throwing exceptions related to this
27: //     program.
28: //
29:
30: class yshell_exn: public runtime_error {
31:    public:
32:       explicit yshell_exn (const string& what);
33: };
34:
35: //
36: // setexecname -
37: //     Sets the static string to be used as an execname.
38: // execname -
39: //     Returns the basename of the executable image, which is used in
40: //     printing error messags.
41: //
42:
43: void execname (const string&);
44: string& execname();
45:
```

```
46:
47: //
48: // want_echo -
49: //     We want to echo all of cin to cout if either cin or cout
50: //     is not a tty.  This helps make batch processing easier by
51: //     making cout look like a terminal session trace.
52: //
53:
54: bool want_echo();
55:
56: //
57: // exit_status -
58: //     A static class for maintaining the exit status.  The default
59: //     status is EXIT_SUCCESS (0), but can be set to another value,
60: //     such as EXIT_FAILURE (1) to indicate that error messages have
61: //     been printed.
62: //
63:
64: class exit_status {
65:    private:
66:       static int status;
67:    public:
68:       static void set (int);
69:       static int get();
70: };
71:
72: //
73: // split -
74: //     Split a string into a wordvec (as defined above).  Any sequence
75: //     of chars in the delimiter string is used as a separator.  To
76: //     Split a pathname, use "/".  To split a shell command, use " ".
77: //
78:
79: wordvec split (const string& line, const string& delimiter);
80:
81: // complain -
82: //     Used for starting error messages.  Sets the exit status to
83: //     EXIT_FAILURE, writes the program name to cerr, and then
84: //     returns the cerr ostream.  Example:
85: //        complain() << filename << ": some problem" << endl;
86: //
87:
88: ostream& complain();
89:
```

```
 90:
 91: //
 92: // operator<< (vector) -
 93: //    An overloaded template operator which allows vectors to be
 94: //    printed out as a single operator, each element separated from
 95: //    the next with spaces.  The item_t must have an output operator
 96: //    defined for it.
 97: //
 98:
 99: template <typename item_t>
100: ostream& operator<< (ostream& out, const vector<item_t>& vec) {
101:    string space = "";
102:    for (const auto& item: vec) {
103:       out << space << item;
104:       space = " ";
105:    }
106:    return out;
107: }
108:
109: #endif
110:
```

```cpp
 1: // $Id: commands.cpp,v 1.11 2014-06-11 13:49:31-07 - - $
 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: command_fn 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 (command_fn)
24:    command_map::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:
```

```cpp
 1: // $Id: debug.cpp,v 1.8 2015-01-02 18:13:23-08 - - $
 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: //
22: // getflag -
23: //    Check to see if a certain flag is on.
24: //
25:
26: bool debugflags::getflag (char flag) {
27:    // WARNING: Don't TRACE this function or the stack will blow up.
28:    return flags.test (static_cast<unsigned char> (flag));
29: }
30:
31: void debugflags::where (char flag, const char* file, int line,
32:                         const char* func) {
33:    cout << execname() << ": DEBUG(" << flag << ") "
34:         << file << "[" << line << "] " << func << "()" << endl;
35: }
36:
```

```cpp
 1: // $Id: inode.cpp,v 1.12 2014-07-03 13:29:57-07 - - $
 2:
 3: #include <iostream>
 4: #include <stdexcept>
 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 PLAIN_INODE:
18:             contents = make_shared<plain_file>();
19:             break;
20:       case DIR_INODE:
21:             contents = make_shared<directory>();
22:             break;
23:    }
24:    DEBUGF ('i', "inode " << inode_nr << ", type = " << type);
25: }
26:
27: int inode::get_inode_nr() const {
28:    DEBUGF ('i', "inode = " << inode_nr);
29:    return inode_nr;
30: }
31:
32: plain_file_ptr plain_file_ptr_of (file_base_ptr ptr) {
33:    plain_file_ptr pfptr = dynamic_pointer_cast<plain_file> (ptr);
34:    if (pfptr == nullptr) throw invalid_argument ("plain_file_ptr_of");
35:    return pfptr;
36: }
37:
38: directory_ptr directory_ptr_of (file_base_ptr ptr) {
39:    directory_ptr dirptr = dynamic_pointer_cast<directory> (ptr);
40:    if (dirptr == nullptr) throw invalid_argument ("directory_ptr_of");
41:    return dirptr;
42: }
43:
```

```
44:
45: size_t plain_file::size() const {
46:    size_t size {0};
47:    DEBUGF ('i', "size = " << size);
48:    return size;
49: }
50:
51: const wordvec& plain_file::readfile() const {
52:    DEBUGF ('i', data);
53:    return data;
54: }
55:
56: void plain_file::writefile (const wordvec& words) {
57:    DEBUGF ('i', words);
58: }
59:
60: size_t directory::size() const {
61:    size_t size {0};
62:    DEBUGF ('i', "size = " << size);
63:    return size;
64: }
65:
66: void directory::remove (const string& filename) {
67:    DEBUGF ('i', filename);
68: }
69:
70: inode_state::inode_state() {
71:    DEBUGF ('i', "root = " << root << ", cwd = " << cwd
72:            << ", prompt = \"" << prompt << "\"");
73: }
74:
75: ostream& operator<< (ostream& out, const inode_state& state) {
76:    out << "inode_state: root = " << state.root
77:        << ", cwd = " << state.cwd;
78:    return out;
79: }
80:
```

```cpp
 1: // $Id: util.cpp,v 1.10 2014-06-11 13:34:25-07 - - $
 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 execname (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:     constexpr int CIN_FD {0};
36:     constexpr int COUT_FD {1};
37:     bool cin_is_not_a_tty = not isatty (CIN_FD);
38:     bool cout_is_not_a_tty = not isatty (COUT_FD);
39:     DEBUGF ('u', "cin_is_not_a_tty = " << cin_is_not_a_tty
40:             << ", cout_is_not_a_tty = " << cout_is_not_a_tty);
41:     return cin_is_not_a_tty or cout_is_not_a_tty;
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:
```

```cpp
 1: // $Id: main.cpp,v 1.3 2014-06-11 13:52:31-07 - - $
 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:    execname (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:             command_fn 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 1.14 2014-06-25 17:42:27-07 - - $
 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++ -g -O0 -Wall -Wextra -rdynamic -std=gnu++11
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
14: EXECBIN      = yshell
15: OBJECTS      = ${CPPSOURCE:.cpp=.o}
16: OTHERS       = ${MKFILE} README
17: ALLSOURCES   = ${CPPHEADER} ${CPPSOURCE} ${OTHERS}
18: LISTING      = Listing.ps
19:
20: all : ${EXECBIN}
21:         - checksource ${ALLSOURCES}
22:
23: ${EXECBIN} : ${OBJECTS}
24:         ${COMPILECPP} -o $@ ${OBJECTS}
25:
26: %.o : %.cpp
27:         ${COMPILECPP} -c $<
28:
29: ci : ${ALLSOURCES}
30:         cid + ${ALLSOURCES}
31:         - checksource ${ALLSOURCES}
32:
33: lis : ${ALLSOURCES}
34:         mkpspdf ${LISTING} ${ALLSOURCES} ${DEPFILE}
35:
36: clean :
37:         - rm ${OBJECTS} ${DEPFILE} core ${EXECBIN}.errs
38:
39: spotless : clean
40:         - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
41:
42: dep : ${CPPSOURCE} ${CPPHEADER}
43:         @ echo "# ${DEPFILE} created `LC_TIME=C date`" >${DEPFILE}
44:         ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPFILE}
45:
46: ${DEPFILE} : ${MKFILE}
47:         @ touch ${DEPFILE}
48:         ${GMAKE} dep
49:
50: again :
51:         ${GMAKE} spotless dep ci all lis
52:
53: ifeq (${NEEDINCL}, )
54: include ${DEPFILE}
55: endif
56:
```

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

03/25/15
19:10:58

$cmps109-wm/Assignments/asg1-shell-fnptrs/code/
README

1/1

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

```
1: # Makefile.dep created Wed Mar 25 19:11:03 PDT 2015
2: commands.o: commands.cpp commands.h inode.h util.h debug.h
3: debug.o: debug.cpp debug.h util.h
4: inode.o: inode.cpp debug.h inode.h util.h
5: util.o: util.cpp util.h debug.h
6: main.o: main.cpp commands.h inode.h util.h debug.h
```