```
 1: // $Id: commands.h,v 1.12 2021-12-20 12:55:34-08 - - $
 2:
 3: #ifndef COMMANDS_H
 4: #define COMMANDS_H
 5:
 6: #include <unordered_map>
 7: using namespace std;
 8:
 9: #include "file_sys.h"
10: #include "util.h"
11:
12: // A couple of convenient usings to avoid verbosity.
13:
14: using command_fn = void (*)(inode_state& state, const wordvec& words);
15: using command_hash = unordered_map<string,command_fn>;
16:
17: // command_error -
18: //    Extend runtime_error for throwing exceptions related to this
19: //    program.
20:
21: class command_error: public runtime_error {
22:    public:
23:       explicit command_error (const string& what);
24: };
25:
26: // execution functions -
27:
28: void fn_cat    (inode_state& state, const wordvec& words);
29: void fn_cd     (inode_state& state, const wordvec& words);
30: void fn_echo   (inode_state& state, const wordvec& words);
31: void fn_exit   (inode_state& state, const wordvec& words);
32: void fn_ls     (inode_state& state, const wordvec& words);
33: void fn_lsr    (inode_state& state, const wordvec& words);
34: void fn_make   (inode_state& state, const wordvec& words);
35: void fn_mkdir  (inode_state& state, const wordvec& words);
36: void fn_prompt (inode_state& state, const wordvec& words);
37: void fn_pwd    (inode_state& state, const wordvec& words);
38: void fn_rm     (inode_state& state, const wordvec& words);
39: void fn_rmr    (inode_state& state, const wordvec& words);
40:
41: command_fn find_command_fn (const string& command);
42:
43: // exit_status_message -
44: //    Prints an exit message and returns the exit status, as recorded
45: //    by any of the functions.
46:
47: int exit_status_message();
48: class ysh_exit: public exception {};
49:
50: #endif
51:
```

```
 1: // $Id: commands.cpp,v 1.23 2022-01-14 18:48:50-08 - - $
 2:
 3: #include "commands.h"
 4: #include "debug.h"
 5:
 6: const command_hash cmd_hash {
 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:    {"rmr"   , fn_rmr   },
19: };
20:
21: command_fn find_command_fn (const string& cmd) {
22:    // Note: value_type is pair<const key_type, mapped_type>
23:    // So: iterator->first is key_type (string)
24:    // So: iterator->second is mapped_type (command_fn)
25:    DEBUGF ('c', "[" << cmd << "]");
26:    const auto result {cmd_hash.find (cmd)};
27:    if (result == cmd_hash.end()) {
28:       throw command_error (cmd + ": no such command");
29:    }
30:    return result->second;
31: }
32:
33: command_error::command_error (const string& what):
34:             runtime_error (what) {
35: }
36:
37: int exit_status_message() {
38:    int status {exec::status()};
39:    cout << exec::execname() << ": exit(" << status << ")" << endl;
40:    return status;
41: }
42:
```

```
43:
44: void fn_cat (inode_state& state, const wordvec& words) {
45:    DEBUGF ('c', state);
46:    DEBUGF ('c', words);
47: }
48:
49: void fn_cd (inode_state& state, const wordvec& words) {
50:    DEBUGF ('c', state);
51:    DEBUGF ('c', words);
52: }
53:
54: void fn_echo (inode_state& state, const wordvec& words) {
55:    DEBUGF ('c', state);
56:    DEBUGF ('c', words);
57:    cout << word_range (words.cbegin() + 1, words.cend()) << endl;
58: }
59:
60: void fn_exit (inode_state& state, const wordvec& words) {
61:    DEBUGF ('c', state);
62:    DEBUGF ('c', words);
63:    throw ysh_exit();
64: }
65:
66: void fn_ls (inode_state& state, const wordvec& words) {
67:    DEBUGF ('c', state);
68:    DEBUGF ('c', words);
69: }
70:
71: void fn_lsr (inode_state& state, const wordvec& words) {
72:    DEBUGF ('c', state);
73:    DEBUGF ('c', words);
74: }
75:
```

```
 76:
 77: void fn_make (inode_state& state, const wordvec& words) {
 78:    DEBUGF ('c', state);
 79:    DEBUGF ('c', words);
 80: }
 81:
 82: void fn_mkdir (inode_state& state, const wordvec& words) {
 83:    DEBUGF ('c', state);
 84:    DEBUGF ('c', words);
 85: }
 86:
 87: void fn_prompt (inode_state& state, const wordvec& words) {
 88:    DEBUGF ('c', state);
 89:    DEBUGF ('c', words);
 90: }
 91:
 92: void fn_pwd (inode_state& state, const wordvec& words) {
 93:    DEBUGF ('c', state);
 94:    DEBUGF ('c', words);
 95: }
 96:
 97: void fn_rm (inode_state& state, const wordvec& words) {
 98:    DEBUGF ('c', state);
 99:    DEBUGF ('c', words);
100: }
101:
102: void fn_rmr (inode_state& state, const wordvec& words) {
103:    DEBUGF ('c', state);
104:    DEBUGF ('c', words);
105: }
106:
```

```
 1: // $Id: debug.h,v 1.13 2021-12-20 12:55:34-08 - - $
 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 trace code.  First argument is a
34: //    trace 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 1.16 2021-10-29 21:18:11-07 - - $
 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: // getflag -
22: //    Check to see if a certain flag is on.
23:
24: bool debugflags::getflag (char flag) {
25:    // WARNING: Don't TRACE this function or the stack will blow up.
26:    return flags_.test (static_cast<unsigned char> (flag));
27: }
28:
29: void debugflags::where (char flag, const char* file, int line,
30:                         const char* pretty_function) {
31:    cerr << "DEBUG(" << flag << ") "
32:         << file << "[" << line << "] " << endl
33:         << "... " << pretty_function << endl;
34: }
35:
```

```
 1: // $Id: file_sys.h,v 1.11 2021-12-20 12:55:34-08 - - $
 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: // inode_t -
16: //    An inode is either a directory or a plain file.
17:
18: enum class file_type {PLAIN_TYPE, DIRECTORY_TYPE};
19: class inode;
20: class base_file;
21: class plain_file;
22: class directory;
23: using inode_ptr = shared_ptr<inode>;
24: using base_file_ptr = shared_ptr<base_file>;
25: ostream& operator<< (ostream&, file_type);
26:
```

```
27:
28: // inode_state -
29: //    A small convenient class to maintain the state of the simulated
30: //    process:  the root (/), the current directory (.), and the
31: //    prompt.
32:
33: class inode_state {
34:    friend class inode;
35:    friend ostream& operator<< (ostream& out, const inode_state&);
36:    private:
37:       inode_ptr root {nullptr};
38:       inode_ptr cwd {nullptr};
39:       string prompt_ {"% "};
40:    public:
41:       inode_state (const inode_state&) = delete; // copy ctor
42:       inode_state& operator= (const inode_state&) = delete; // op=
43:       inode_state();
44:       const string& prompt() const;
45:       void prompt (const string&);
46: };
47:
48: // class inode -
49: // inode ctor -
50: //    Create a new inode of the given type.
51: // get_inode_nr -
52: //    Retrieves the serial number of the inode.  Inode numbers are
53: //    allocated in sequence by small integer.
54: // size -
55: //    Returns the size of an inode.  For a directory, this is the
56: //    number of dirents.  For a text file, the number of characters
57: //    when printed (the sum of the lengths of each word, plus the
58: //    number of words.
59: //
60:
61: class inode {
62:    friend class inode_state;
63:    private:
64:       static size_t next_inode_nr;
65:       size_t inode_nr;
66:       base_file_ptr contents;
67:    public:
68:       inode() = delete;
69:       inode (const inode&) = delete;
70:       inode& operator= (const inode&) = delete;
71:       inode (file_type);
72:       size_t get_inode_nr() const;
73: };
74:
```

```
 75:
 76: // class base_file -
 77: // Just a base class at which an inode can point.  No data or
 78: // functions.  Makes the synthesized members useable only from
 79: // the derived classes.
 80:
 81: class file_error: public runtime_error {
 82:    public:
 83:       explicit file_error (const string& what);
 84: };
 85:
 86: class base_file {
 87:    protected:
 88:       base_file() = default;
 89:       virtual const string& error_file_type() const = 0;
 90:    public:
 91:       virtual ~base_file() = default;
 92:       base_file (const base_file&) = delete;
 93:       base_file& operator= (const base_file&) = delete;
 94:       virtual size_t size() const = 0;
 95:       virtual const wordvec& readfile() const;
 96:       virtual void writefile (const wordvec& newdata);
 97:       virtual void remove (const string& filename);
 98:       virtual inode_ptr mkdir (const string& dirname);
 99:       virtual inode_ptr mkfile (const string& filename);
100: };
```

```
101:
102: // class plain_file -
103: // Used to hold data.
104: // synthesized default ctor -
105: //    Default vector<string> is a an empty vector.
106: // readfile -
107: //    Returns a copy of the contents of the wordvec in the file.
108: // writefile -
109: //    Replaces the contents of a file with new contents.
110:
111: class plain_file: public base_file {
112:    private:
113:       wordvec data;
114:       virtual const string& error_file_type() const override {
115:          static const string result = "plain file";
116:          return result;
117:       }
118:    public:
119:       virtual size_t size() const override;
120:       virtual const wordvec& readfile() const override;
121:       virtual void writefile (const wordvec& newdata) override;
122: };
123:
124: // class directory -
125: // Used to map filenames onto inode pointers.
126: // default ctor -
127: //    Creates a new map with keys "." and "..".
128: // remove -
129: //    Removes the file or subdirectory from the current inode.
130: //    Throws an file_error if this is not a directory, the file
131: //    does not exist, or the subdirectory is not empty.
132: //    Here empty means the only entries are dot (.) and dotdot (..).
133: // mkdir -
134: //    Creates a new directory under the current directory and
135: //    immediately adds the directories dot (.) and dotdot (..) to it.
136: //    Note that the parent (..) of / is / itself.  It is an error
137: //    if the entry already exists.
138: // mkfile -
139: //    Create a new empty text file with the given name.  Error if
140: //    a dirent with that name exists.
141:
142: class directory: public base_file {
143:    private:
144:       // Must be a map, not unordered_map, so printing is lexicographic
145:       map<string,inode_ptr> dirents;
146:       virtual const string& error_file_type() const override {
147:          static const string result = "directory";
148:          return result;
149:       }
150:    public:
151:       virtual size_t size() const override;
152:       virtual void remove (const string& filename) override;
153:       virtual inode_ptr mkdir (const string& dirname) override;
154:       virtual inode_ptr mkfile (const string& filename) override;
155: };
156:
157: #endif
158:
```

```
 1: // $Id: file_sys.cpp,v 1.11 2022-01-14 18:48:50-08 - - $
 2:
 3: #include <cassert>
 4: #include <iostream>
 5: #include <stdexcept>
 6:
 7: using namespace std;
 8:
 9: #include "debug.h"
10: #include "file_sys.h"
11:
12: size_t inode::next_inode_nr {1};
13:
14: ostream& operator<< (ostream& out, file_type type) {
15:    switch (type) {
16:       case file_type::PLAIN_TYPE: out << "PLAIN_TYPE"; break;
17:       case file_type::DIRECTORY_TYPE: out << "DIRECTORY_TYPE"; break;
18:       default: assert (false);
19:    };
20:    return out;
21: }
22:
23: inode_state::inode_state() {
24:    DEBUGF ('i', "root = " << root << ", cwd = " << cwd
25:           << ", prompt = \"" << prompt() << "\"");
26: }
27:
28: const string& inode_state::prompt() const { return prompt_; }
29:
30: void inode_state::prompt (const string& new_prompt) {
31:    prompt_ = new_prompt;
32: }
33:
34: ostream& operator<< (ostream& out, const inode_state& state) {
35:    out << "inode_state: root = " << state.root
36:        << ", cwd = " << state.cwd;
37:    return out;
38: }
39:
40: inode::inode(file_type type): inode_nr (next_inode_nr++) {
41:    switch (type) {
42:       case file_type::PLAIN_TYPE:
43:            contents = make_shared<plain_file>();
44:            break;
45:       case file_type::DIRECTORY_TYPE:
46:            contents = make_shared<directory>();
47:            break;
48:       default: assert (false);
49:    }
50:    DEBUGF ('i', "inode " << inode_nr << ", type = " << type);
51: }
52:
53: size_t inode::get_inode_nr() const {
54:    DEBUGF ('i', "inode = " << inode_nr);
55:    return inode_nr;
56: }
57:
```

```
58:
59: file_error::file_error (const string& what):
60:             runtime_error (what) {
61: }
62:
63: const wordvec& base_file::readfile() const {
64:    throw file_error ("is a " + error_file_type());
65: }
66:
67: void base_file::writefile (const wordvec&) {
68:    throw file_error ("is a " + error_file_type());
69: }
70:
71: void base_file::remove (const string&) {
72:    throw file_error ("is a " + error_file_type());
73: }
74:
75: inode_ptr base_file::mkdir (const string&) {
76:    throw file_error ("is a " + error_file_type());
77: }
78:
79: inode_ptr base_file::mkfile (const string&) {
80:    throw file_error ("is a " + error_file_type());
81: }
82:
```

```
 83:
 84: size_t plain_file::size() const {
 85:    size_t size {0};
 86:    DEBUGF ('i', "size = " << size);
 87:    return size;
 88: }
 89:
 90: const wordvec& plain_file::readfile() const {
 91:    DEBUGF ('i', data);
 92:    return data;
 93: }
 94:
 95: void plain_file::writefile (const wordvec& words) {
 96:    DEBUGF ('i', words);
 97: }
 98:
 99: size_t directory::size() const {
100:    size_t size {0};
101:    DEBUGF ('i', "size = " << size);
102:    return size;
103: }
104:
105: void directory::remove (const string& filename) {
106:    DEBUGF ('i', filename);
107: }
108:
109: inode_ptr directory::mkdir (const string& dirname) {
110:    DEBUGF ('i', dirname);
111:    return nullptr;
112: }
113:
114: inode_ptr directory::mkfile (const string& filename) {
115:    DEBUGF ('i', filename);
116:    return nullptr;
117: }
118:
```

```
 1: // $Id: util.h,v 1.16 2021-12-20 12:55:34-08 - - $
 2:
 3: // util -
 4: //    A utility to provide various services not conveniently
 5: //    included in other modules.
 6:
 7: #ifndef UTIL_H
 8: #define UTIL_H
 9:
10: #include <iostream>
11: #include <stdexcept>
12: #include <string>
13: #include <vector>
14: using namespace std;
15:
16: // Convenient type using to allow brevity of code elsewhere.
17:
18: template <typename iterator>
19: using range_type = pair<iterator,iterator>;
20:
21: using wordvec = vector<string>;
22: using word_range = range_type<decltype(declval<wordvec>().cbegin())>;
23:
24: // want_echo -
25: //    We want to echo all of cin to cout if either cin or cout
26: //    is not a tty.  This helps make batch processing easier by
27: //    making cout look like a terminal session trace.
28:
29: bool want_echo();
30:
31: //
32: // main -
33: //    Keep track of execname and exit status.  Must be initialized
34: //    as the first thing done inside main.  Main should call:
35: //       main::execname (argv[0]);
36: //    before anything else.
37: //
38:
39: class exec {
40:    private:
41:       static string execname_;
42:       static int status_;
43:       static void execname (const string& argv0);
44:       friend int main (int, char**);
45:    public:
46:       static void status (int status);
47:       static const string& execname() {return execname_; }
48:       static int status() {return status_; }
49: };
50:
```

```
51:
52: // split -
53: //    Split a string into a wordvec (as defined above).  Any sequence
54: //    of chars in the delimiter string is used as a separator.  To
55: //    Split a pathname, use "/".  To split a shell command, use " ".
56:
57: wordvec split (const string& line, const string& delimiter);
58:
59: // complain -
60: //    Used for starting error messages.  Sets the exit status to
61: //    EXIT_FAILURE, writes the program name to cerr, and then
62: //    returns the cerr ostream.  Example:
63: //       complain() << filename << ": some problem" << endl;
64:
65: ostream& complain();
66:
67: // operator<< (vector) -
68: //    An overloaded template operator which allows vectors to be
69: //    printed out as a single operator, each element separated from
70: //    the next with spaces.  The item_t must have an output operator
71: //    defined for it.
72:
73: template <typename item_t>
74: ostream& operator<< (ostream& out, const vector<item_t>& vec) {
75:    string space {""};
76:    for (const auto& item: vec) {
77:       out << space << item;
78:       space = " ";
79:    }
80:    return out;
81: }
82:
83: template <typename iterator>
84: ostream& operator<< (ostream& out, range_type<iterator> range) {
85:    for (auto itor = range.first; itor != range.second; ++itor) {
86:       if (itor != range.first) out << " ";
87:       out << *itor;
88:    }
89:    return out;
90: }
91:
92: #endif
93:
```

```
 1: // $Id: util.cpp,v 1.15 2021-09-26 12:41:17-07 - - $
 2:
 3: #include <cstdlib>
 4: #include <unistd.h>
 5:
 6: using namespace std;
 7:
 8: #include "util.h"
 9: #include "debug.h"
10:
11: bool want_echo() {
12:    constexpr int CIN_FD {0};
13:    constexpr int COUT_FD {1};
14:    bool cin_is_not_a_tty = not isatty (CIN_FD);
15:    bool cout_is_not_a_tty = not isatty (COUT_FD);
16:    DEBUGF ('u', "cin_is_not_a_tty = " << cin_is_not_a_tty
17:           << ", cout_is_not_a_tty = " << cout_is_not_a_tty);
18:    return cin_is_not_a_tty or cout_is_not_a_tty;
19: }
20:
21: string exec::execname_; // Must be initialized from main().
22: int exec::status_ {EXIT_SUCCESS};
23:
24: string basename (const string &arg) {
25:    return arg.substr (arg.find_last_of ('/') + 1);
26: }
27:
28: void exec::execname (const string& argv0) {
29:    execname_ = basename (argv0);
30:    cout << boolalpha;
31:    cerr << boolalpha;
32:    DEBUGF ('u', "execname = " << execname_);
33: }
34:
35: void exec::status (int status) {
36:    if (status_ < status) status_ = status;
37: }
38:
```

```
39:
40: wordvec split (const string& line, const string& delimiters) {
41:    wordvec words;
42:    size_t end {0};
43:
44:    // Loop over the string, splitting out words, and for each word
45:    // thus found, append it to the output wordvec.
46:    for (;;) {
47:       size_t start {line.find_first_not_of (delimiters, end)};
48:       if (start == string::npos) break;
49:       end = line.find_first_of (delimiters, start);
50:       words.push_back (line.substr (start, end - start));
51:    }
52:    DEBUGF ('u', words);
53:    return words;
54: }
55:
56: ostream& complain() {
57:    exec::status (EXIT_FAILURE);
58:    cerr << exec::execname() << ": ";
59:    return cerr;
60: }
61:
```

```
 1: // $Id: main.cpp,v 1.12 2021-09-26 12:41:17-07 - - $
 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 "file_sys.h"
14: #include "util.h"
15:
16: // scan_options
17: //    Options analysis:  The only option is -Dflags.
18:
19: void scan_options (int argc, char** argv) {
20:    opterr = 0;
21:    for (;;) {
22:       int option {getopt (argc, argv, "@:")};
23:       if (option == EOF) break;
24:       switch (option) {
25:          case '@':
26:             debugflags::setflags (optarg);
27:             break;
28:          default:
29:             complain() << "-" << static_cast<char> (option)
30:                        << ": invalid option" << endl;
31:             break;
32:       }
33:    }
34:    if (optind < argc) {
35:       complain() << "operands not permitted" << endl;
36:    }
37: }
38:
```

```
39:
40: // main -
41: //    Main program which loops reading commands until end of file.
42:
43: int main (int argc, char** argv) {
44:    exec::execname (argv[0]);
45:    cout << boolalpha;  // Print false or true instead of 0 or 1.
46:    cerr << boolalpha;
47:    cout << argv[0] << " build " << __DATE__ << " " << __TIME__ << endl;
48:    scan_options (argc, argv);
49:    bool need_echo {want_echo()};
50:    inode_state state;
51:    try {
52:       for (;;) {
53:          try {
54:             // Read a line, break at EOF, and echo print the prompt
55:             // if one is needed.
56:             cout << state.prompt();
57:             string line;
58:             getline (cin, line);
59:             if (cin.eof()) {
60:                if (need_echo) cout << "^D";
61:                cout << endl;
62:                DEBUGF ('y', "EOF");
63:                break;
64:             }
65:             if (need_echo) cout << line << endl;
66:
67:             // Split the line into words and lookup the appropriate
68:             // function.  Complain or call it.
69:             wordvec words = split (line, " \t");
70:             DEBUGF ('y', "words = " << words);
71:             command_fn fn = find_command_fn (words.at(0));
72:             fn (state, words);
73:          }catch (file_error& error) {
74:             complain() << error.what() << endl;
75:          }catch (command_error& error) {
76:             complain() << error.what() << endl;
77:          }
78:       }
79:    } catch (ysh_exit&) {
80:       // This catch intentionally left blank.
81:    }
82:
83:    return exit_status_message();
84: }
85:
```

```
 1: # $Id: Makefile,v 1.41 2021-09-26 12:41:17-07 - - $
 2:
 3: MKFILE      = Makefile
 4: DEPFILE     = ${MKFILE}.dep
 5: NOINCL      = check lint ci clean spotless
 6: NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
 7: GMAKE       = ${MAKE} --no-print-directory
 8: GPPWARN     = -Wall -Wextra -Wpedantic -Wshadow -Wold-style-cast
 9: GPPOPTS     = ${GPPWARN} -fdiagnostics-color=never
10: COMPILECPP  = g++ -std=gnu++2a -g -O0 ${GPPOPTS}
11: MAKEDEPCPP  = g++ -std=gnu++2a -MM ${GPPOPTS}
12:
13: MODULES     = commands debug file_sys util
14: CPPHEADER   = ${MODULES:=.h}
15: CPPSOURCE   = ${MODULES:=.cpp} main.cpp
16: EXECBIN     = yshell
17: OBJECTS     = ${CPPSOURCE:.cpp=.o}
18: MODULESRC   = ${foreach MOD, ${MODULES}, ${MOD}.h ${MOD}.cpp}
19: OTHERSRC    = ${filter-out ${MODULESRC}, ${CPPHEADER} ${CPPSOURCE}}
20: ALLSOURCES  = ${MODULESRC} ${OTHERSRC} ${MKFILE}
21: LISTING     = Listing.ps
22:
23: export PATH := ${PATH}:/afs/cats.ucsc.edu/courses/cse110a-wm/bin
24:
25: all : ${EXECBIN}
26:
27: ${EXECBIN} : ${OBJECTS}
28:         ${COMPILECPP} -o $@ ${OBJECTS}
29:
30: %.o : %.cpp
31:         - checksource $<
32:         - cpplint.py.perl $<
33:         ${COMPILECPP} -c $<
34:
35: ci : check
36:         - cid -is ${ALLSOURCES}
37:
38: check : ${ALLSOURCES}
39:         - checksource ${ALLSOURCES}
40:         - cpplint.py.perl ${CPPSOURCE}
41:
42: lis : ${ALLSOURCES}
43:         mkpspdf ${LISTING} ${ALLSOURCES} ${DEPFILE}
44:
45: clean :
46:         - rm ${OBJECTS} ${DEPFILE} core ${EXECBIN}.errs
47:
48: spotless : clean
49:         - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
50:
```

```
51:
52: dep : ${CPPSOURCE} ${CPPHEADER}
53:         @ echo "# ${DEPFILE} created `LC_TIME=C date`" >${DEPFILE}
54:         ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPFILE}
55:
56: ${DEPFILE} : ${MKFILE}
57:         @ touch ${DEPFILE}
58:         ${GMAKE} dep
59:
60: again :
61:         ${GMAKE} spotless dep ci all lis
62:
63: ifeq (${NEEDINCL}, )
64: include ${DEPFILE}
65: endif
66:
```

```
1: # Makefile.dep created Fri Jan 14 18:51:00 PST 2022
2: commands.o: commands.cpp commands.h file_sys.h util.h debug.h
3: debug.o: debug.cpp debug.h util.h
4: file_sys.o: file_sys.cpp debug.h file_sys.h util.h
5: util.o: util.cpp util.h debug.h
6: main.o: main.cpp commands.h file_sys.h util.h debug.h
```