```
 1: // $Id: commands.h,v 1.11 2016-01-14 14:45:21-08 - - $
 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 1.17 2018-01-25 14:02:55-08 - - $
 2:
 3: #include "commands.h"
 4: #include "debug.h"
 5:
 6: 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: };
19:
20: command_fn find_command_fn (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:    DEBUGF ('c', "[" << cmd << "]");
25:    const auto result = cmd_hash.find (cmd);
26:    if (result == cmd_hash.end()) {
27:       throw command_error (cmd + ": no such function");
28:    }
29:    return result->second;
30: }
31:
32: command_error::command_error (const string& what):
33:             runtime_error (what) {
34: }
35:
36: int exit_status_message() {
37:    int exit_status = exit_status::get();
38:    cout << execname() << ": exit(" << exit_status << ")" << endl;
39:    return exit_status;
40: }
41:
42: void fn_cat (inode_state& state, const wordvec& words){
43:    DEBUGF ('c', state);
44:    DEBUGF ('c', words);
45: }
46:
47: void fn_cd (inode_state& state, const wordvec& words){
48:    DEBUGF ('c', state);
49:    DEBUGF ('c', words);
50: }
51:
52: void fn_echo (inode_state& state, const wordvec& words){
53:    DEBUGF ('c', state);
54:    DEBUGF ('c', words);
55:    cout << word_range (words.cbegin() + 1, words.cend()) << endl;
56: }
57:
```

```
 58:
 59: void fn_exit (inode_state& state, const wordvec& words){
 60:    DEBUGF ('c', state);
 61:    DEBUGF ('c', words);
 62:    throw ysh_exit();
 63: }
 64:
 65: void fn_ls (inode_state& state, const wordvec& words){
 66:    DEBUGF ('c', state);
 67:    DEBUGF ('c', words);
 68: }
 69:
 70: void fn_lsr (inode_state& state, const wordvec& words){
 71:    DEBUGF ('c', state);
 72:    DEBUGF ('c', words);
 73: }
 74:
 75: void fn_make (inode_state& state, const wordvec& words){
 76:    DEBUGF ('c', state);
 77:    DEBUGF ('c', words);
 78: }
 79:
 80: void fn_mkdir (inode_state& state, const wordvec& words){
 81:    DEBUGF ('c', state);
 82:    DEBUGF ('c', words);
 83: }
 84:
 85: void fn_prompt (inode_state& state, const wordvec& words){
 86:    DEBUGF ('c', state);
 87:    DEBUGF ('c', words);
 88: }
 89:
 90: void fn_pwd (inode_state& state, const wordvec& words){
 91:    DEBUGF ('c', state);
 92:    DEBUGF ('c', words);
 93: }
 94:
 95: void fn_rm (inode_state& state, const wordvec& words){
 96:    DEBUGF ('c', state);
 97:    DEBUGF ('c', words);
 98: }
 99:
100: void fn_rmr (inode_state& state, const wordvec& words){
101:    DEBUGF ('c', state);
102:    DEBUGF ('c', words);
103: }
104:
```

```
 1: // $Id: debug.h,v 1.10 2018-01-25 14:02:55-08 - - $
 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, each indicated
13: //    by a single character.
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 trace code.  First argument is a
35: //    trace 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 DEBUGF(FLAG,CODE) ;
44: #define DEBUGS(FLAG,STMT) ;
45: #else
46: #define DEBUGF(FLAG,CODE) { \
47:            if (debugflags::getflag (FLAG)) { \
48:               debugflags::where (FLAG, __FILE__, __LINE__, \
49:                                  __PRETTY_FUNCTION__); \
50:               cerr << CODE << endl; \
51:            } \
52:         }
53: #define DEBUGS(FLAG,STMT) { \
54:            if (debugflags::getflag (FLAG)) { \
55:               debugflags::where (FLAG, __FILE__, __LINE__, \
56:                                  __PRETTY_FUNCTION__); \
57:               STMT; \
58:            } \
59:         }
60: #endif
61:
62: #endif
63:
```

```
 1: // $Id: debug.cpp,v 1.12 2018-06-27 14:44:57-07 - - $
 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:    cout << execname() << ": DEBUG(" << flag << ") "
32:         << file << "[" << line << "] " << endl
33:         << "   " << pretty_function << endl;
34: }
35:
```

```
 1: // $Id: file_sys.h,v 1.6 2018-06-27 14:44:57-07 - - $
 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: // 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: };
46:
47: // class inode -
48: // inode ctor -
49: //    Create a new inode of the given type.
50: // get_inode_nr -
51: //    Retrieves the serial number of the inode.  Inode numbers are
52: //    allocated in sequence by small integer.
53: // size -
54: //    Returns the size of an inode.  For a directory, this is the
55: //    number of dirents.  For a text file, the number of characters
56: //    when printed (the sum of the lengths of each word, plus the
57: //    number of words.
58: //
59:
60: class inode {
61:    friend class inode_state;
62:    private:
63:       static int next_inode_nr;
64:       int inode_nr;
65:       base_file_ptr contents;
66:    public:
67:       inode (file_type);
68:       int get_inode_nr() const;
69: };
70:
```

```
71:
72: // class base_file -
73: // Just a base class at which an inode can point.  No data or
74: // functions.  Makes the synthesized members useable only from
75: // the derived classes.
76:
77: class file_error: public runtime_error {
78:    public:
79:       explicit file_error (const string& what);
80: };
81:
82: class base_file {
83:    protected:
84:       base_file() = default;
85:    public:
86:       virtual ~base_file() = default;
87:       base_file (const base_file&) = delete;
88:       base_file& operator= (const base_file&) = delete;
89:       virtual size_t size() const = 0;
90:       virtual const wordvec& readfile() const = 0;
91:       virtual void writefile (const wordvec& newdata) = 0;
92:       virtual void remove (const string& filename) = 0;
93:       virtual inode_ptr mkdir (const string& dirname) = 0;
94:       virtual inode_ptr mkfile (const string& filename) = 0;
95: };
```

```
 96:
 97: // class plain_file -
 98: // Used to hold data.
 99: // synthesized default ctor -
100: //    Default vector<string> is a an empty vector.
101: // readfile -
102: //    Returns a copy of the contents of the wordvec in the file.
103: // writefile -
104: //    Replaces the contents of a file with new contents.
105:
106: class plain_file: public base_file {
107:    private:
108:       wordvec data;
109:    public:
110:       virtual size_t size() const override;
111:       virtual const wordvec& readfile() const override;
112:       virtual void writefile (const wordvec& newdata) override;
113:       virtual void remove (const string& filename) override;
114:       virtual inode_ptr mkdir (const string& dirname) override;
115:       virtual inode_ptr mkfile (const string& filename) override;
116: };
117:
118: // class directory -
119: // Used to map filenames onto inode pointers.
120: // default ctor -
121: //    Creates a new map with keys "." and "..".
122: // remove -
123: //    Removes the file or subdirectory from the current inode.
124: //    Throws an file_error if this is not a directory, the file
125: //    does not exist, or the subdirectory is not empty.
126: //    Here empty means the only entries are dot (.) and dotdot (..).
127: // mkdir -
128: //    Creates a new directory under the current directory and
129: //    immediately adds the directories dot (.) and dotdot (..) to it.
130: //    Note that the parent (..) of / is / itself.  It is an error
131: //    if the entry already exists.
132: // mkfile -
133: //    Create a new empty text file with the given name.  Error if
134: //    a dirent with that name exists.
135:
136: class directory: public base_file {
137:    private:
138:       // Must be a map, not unordered_map, so printing is lexicographic
139:       map<string,inode_ptr> dirents;
140:    public:
141:       virtual size_t size() const override;
142:       virtual const wordvec& readfile() const override;
143:       virtual void writefile (const wordvec& newdata) override;
144:       virtual void remove (const string& filename) override;
145:       virtual inode_ptr mkdir (const string& dirname) override;
146:       virtual inode_ptr mkfile (const string& filename) override;
147: };
148:
149: #endif
150:
```

```
 1: // $Id: file_sys.cpp,v 1.6 2018-06-27 14:44:57-07 - - $
 2:
 3: #include <iostream>
 4: #include <stdexcept>
 5: #include <unordered_map>
 6:
 7: using namespace std;
 8:
 9: #include "debug.h"
10: #include "file_sys.h"
11:
12: int inode::next_inode_nr {1};
13:
14: struct file_type_hash {
15:    size_t operator() (file_type type) const {
16:       return static_cast<size_t> (type);
17:    }
18: };
19:
20: ostream& operator<< (ostream& out, file_type type) {
21:    static unordered_map<file_type,string,file_type_hash> hash {
22:       {file_type::PLAIN_TYPE, "PLAIN_TYPE"},
23:       {file_type::DIRECTORY_TYPE, "DIRECTORY_TYPE"},
24:    };
25:    return out << hash[type];
26: }
27:
28: inode_state::inode_state() {
29:    DEBUGF ('i', "root = " << root << ", cwd = " << cwd
30:           << ", prompt = \"" << prompt() << "\"");
31: }
32:
33: const string& inode_state::prompt() const { return prompt_; }
34:
35: ostream& operator<< (ostream& out, const inode_state& state) {
36:    out << "inode_state: root = " << state.root
37:        << ", cwd = " << state.cwd;
38:    return out;
39: }
40:
41: inode::inode(file_type type): inode_nr (next_inode_nr++) {
42:    switch (type) {
43:       case file_type::PLAIN_TYPE:
44:            contents = make_shared<plain_file>();
45:            break;
46:       case file_type::DIRECTORY_TYPE:
47:            contents = make_shared<directory>();
48:            break;
49:    }
50:    DEBUGF ('i', "inode " << inode_nr << ", type = " << type);
51: }
52:
53: int 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: size_t plain_file::size() const {
64:    size_t size {0};
65:    DEBUGF ('i', "size = " << size);
66:    return size;
67: }
68:
69: const wordvec& plain_file::readfile() const {
70:    DEBUGF ('i', data);
71:    return data;
72: }
73:
74: void plain_file::writefile (const wordvec& words) {
75:    DEBUGF ('i', words);
76: }
77:
78: void plain_file::remove (const string&) {
79:    throw file_error ("is a plain file");
80: }
81:
82: inode_ptr plain_file::mkdir (const string&) {
83:    throw file_error ("is a plain file");
84: }
85:
86: inode_ptr plain_file::mkfile (const string&) {
87:    throw file_error ("is a plain file");
88: }
89:
```

```
 90:
 91: size_t directory::size() const {
 92:    size_t size {0};
 93:    DEBUGF ('i', "size = " << size);
 94:    return size;
 95: }
 96:
 97: const wordvec& directory::readfile() const {
 98:    throw file_error ("is a directory");
 99: }
100:
101: void directory::writefile (const wordvec&) {
102:    throw file_error ("is a directory");
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 1.12 2016-01-14 16:16:52-08 - - $
 2:
 3: // util -
 4: //    A utility class 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: // setexecname -
25: //    Sets the static string to be used as an execname.
26: // execname -
27: //    Returns the basename of the executable image, which is used in
28: //    printing error messags.
29:
30: void execname (const string&);
31: string& execname();
32:
33: // want_echo -
34: //    We want to echo all of cin to cout if either cin or cout
35: //    is not a tty.  This helps make batch processing easier by
36: //    making cout look like a terminal session trace.
37:
38: bool want_echo();
39:
40: // exit_status -
41: //    A static class for maintaining the exit status.  The default
42: //    status is EXIT_SUCCESS (0), but can be set to another value,
43: //    such as EXIT_FAILURE (1) to indicate that error messages have
44: //    been printed.
45:
46: class exit_status {
47:    private:
48:       static int status;
49:    public:
50:       static void set (int);
51:       static int get();
52: };
53:
```

```
54:
55: // split -
56: //    Split a string into a wordvec (as defined above).  Any sequence
57: //    of chars in the delimiter string is used as a separator.  To
58: //    Split a pathname, use "/".  To split a shell command, use " ".
59:
60: wordvec split (const string& line, const string& delimiter);
61:
62: // complain -
63: //    Used for starting error messages.  Sets the exit status to
64: //    EXIT_FAILURE, writes the program name to cerr, and then
65: //    returns the cerr ostream.  Example:
66: //       complain() << filename << ": some problem" << endl;
67:
68: ostream& complain();
69:
70: // operator<< (vector) -
71: //    An overloaded template operator which allows vectors to be
72: //    printed out as a single operator, each element separated from
73: //    the next with spaces.  The item_t must have an output operator
74: //    defined for it.
75:
76: template <typename item_t>
77: ostream& operator<< (ostream& out, const vector<item_t>& vec) {
78:    string space = "";
79:    for (const auto& item: vec) {
80:       out << space << item;
81:       space = " ";
82:    }
83:    return out;
84: }
85:
86: template <typename iterator>
87: ostream& operator<< (ostream& out, range_type<iterator> range) {
88:    for (auto itor = range.first; itor != range.second; ++itor) {
89:       if (itor != range.first) out << " ";
90:       out << *itor;
91:    }
92:    return out;
93: }
94:
95: #endif
96:
```

```
 1: // $Id: util.cpp,v 1.11 2016-01-13 16:21:53-08 - - $
 2:
 3: #include <cstdlib>
 4: #include <unistd.h>
 5:
 6: using namespace std;
 7:
 8: #include "util.h"
 9: #include "debug.h"
10:
11: int exit_status::status = EXIT_SUCCESS;
12: static string execname_string;
13:
14: void exit_status::set (int new_status) {
15:    status = new_status;
16: }
17:
18: int exit_status::get() {
19:    return status;
20: }
21:
22: void execname (const string& name) {
23:    execname_string =  name.substr (name.rfind ('/') + 1);
24:    DEBUGF ('u', execname_string);
25: }
26:
27: string& execname() {
28:    return execname_string;
29: }
30:
31: bool want_echo() {
32:    constexpr int CIN_FD {0};
33:    constexpr int COUT_FD {1};
34:    bool cin_is_not_a_tty = not isatty (CIN_FD);
35:    bool cout_is_not_a_tty = not isatty (COUT_FD);
36:    DEBUGF ('u', "cin_is_not_a_tty = " << cin_is_not_a_tty
37:           << ", cout_is_not_a_tty = " << cout_is_not_a_tty);
38:    return cin_is_not_a_tty or cout_is_not_a_tty;
39: }
40:
```

```
41:
42: wordvec split (const string& line, const string& delimiters) {
43:    wordvec words;
44:    size_t end = 0;
45:
46:    // Loop over the string, splitting out words, and for each word
47:    // thus found, append it to the output wordvec.
48:    for (;;) {
49:       size_t start = line.find_first_not_of (delimiters, end);
50:       if (start == string::npos) break;
51:       end = line.find_first_of (delimiters, start);
52:       words.push_back (line.substr (start, end - start));
53:    }
54:    DEBUGF ('u', words);
55:    return words;
56: }
57:
58: ostream& complain() {
59:    exit_status::set (EXIT_FAILURE);
60:    cerr << execname() << ": ";
61:    return cerr;
62: }
63:
```

```
 1: // $Id: main.cpp,v 1.9 2016-01-14 16:16:52-08 - - $
 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:    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 (command_error& error) {
74:             // If there is a problem discovered in any function, an
75:             // exn is thrown and printed here.
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 1.28 2018-06-27 14:46:28-07 - - $
 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: GPPOPTS     = -Wall -Wextra -Wold-style-cast -fdiagnostics-color=never
 9: COMPILECPP  = g++ -std=gnu++17 -g -O0 ${GPPOPTS}
10: MAKEDEPCPP  = g++ -std=gnu++17 -MM ${GPPOPTS}
11: UTILBIN     = /afs/cats.ucsc.edu/courses/cmps109-wm/bin
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: all : ${EXECBIN}
24:
25: ${EXECBIN} : ${OBJECTS}
26:         ${COMPILECPP} -o $@ ${OBJECTS}
27:
28: %.o : %.cpp
29:         - ${UTILBIN}/cpplint.py.perl $<
30:         - ${UTILBIN}/checksource $<
31:         ${COMPILECPP} -c $<
32:
33: ci : ${ALLSOURCES}
34:         ${UTILBIN}/cid + ${ALLSOURCES}
35:         - ${UTILBIN}/checksource ${ALLSOURCES}
36:
37: lis : ${ALLSOURCES}
38:         mkpspdf ${LISTING} ${ALLSOURCES} ${DEPFILE}
39:
40: clean :
41:         - rm ${OBJECTS} ${DEPFILE} core ${EXECBIN}.errs
42:
43: spotless : clean
44:         - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
45:
```

```
46:
47: dep : ${CPPSOURCE} ${CPPHEADER}
48:         @ echo "# ${DEPFILE} created `LC_TIME=C date`" >${DEPFILE}
49:         ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPFILE}
50:
51: ${DEPFILE} : ${MKFILE}
52:         @ touch ${DEPFILE}
53:         ${GMAKE} dep
54:
55: again :
56:         ${GMAKE} spotless dep ci all lis
57:
58: ifeq (${NEEDINCL}, )
59: include ${DEPFILE}
60: endif
61:
```

```
1: # Makefile.dep created Wed Jun 27 14:46:27 PDT 2018
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
```