

```
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 1.22 2021-05-03 17:36:57-07 - - $
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:    {"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 function");
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: void fn_cat (inode_state& state, const wordvec& words) {
44:     DEBUGF ('c', state);
45:     DEBUGF ('c', words);
46:     //The contents of each file is copied to the standard output. An error
47:     //reported if no files are specified, a file does not exist, or is a
48:     //directory.
49:     if(words.size() == 1){ //if no files are specified
50:         throw command_error ("cat: No files are specified"); //dont work
51:         // program needs to continue!!!!!!
52:     }
53:     else{
54:         wordvec split_path;
55:         shared_ptr <directory> state_dir = dynamic_pointer_cast<director
y>
        (state.get_cwd()->get_contents());
```

```
56:         // int jcount = 1;
57:         for(unsigned long j = 1; j < words.size(); ++j){
58:             //if there is a path, but checks on each call
59:             split_path = split(words.at(j), "/"); //skips make call but inclu
des all paths
60:             // cout<< "split_path:";
61:             // cout<< split_path;
62:             // cout<< "end split path";
63:             int count = 0;
64:             if(split_path.size() > 1) { //if its a path
65:                 for(auto i = split_path.begin(); i < split_path.end() - 1; i++) {
//skip last cause creating last so wont exist
66:                     if(state_dir->file_dne(split_path.at(count)) == true){
67:                         throw command_error ("cat: "+split_path.at(count)+" : N
o such file or directory");
68:                         //like foo/wrongdir/bar/newfile
69:                         //would output wrongdir: no such file or dir
70:                     }
71:                     else if(state_dir->is_dir_(split_path.at(count)) == false
){
72:                         throw command_error ("cat: "+split_path.at(count)+" : N
o such file or directory");
73:                     }
74:                     count++; //cause the auto is an itr
75:                 }
76:                 //file needs to exist
77:                 if(state_dir->file_dne(words.at(count)) == true){
78:                     throw command_error("cat: "+ words.at(j) +": No such file
or directory");
79:                 }
80:                 //and not be a directory
81:                 else if(state_dir->is_dir_(words.at(count)) == true){
82:                     throw command_error("cat: "+ words.at(j) +": Is a directo
ry");
83:                 }
84:
85:                 cout<< state_dir->get_second(words.at(count))->get_contents(
)->readfile(); //this works
86:                 cout<< "\n";
87:             } //endif
88:             //if no path
89:             else{
90:                 if(state_dir->file_dne(words.at(j)) == true){
91:                     throw command_error("cat: "+ words.at(j) +": No such file
or directory");
92:                 }
93:                 else if(state_dir->is_dir_(words.at(j)) == true){
94:                     throw command_error("cat: "+ words.at(j) +": Is a directo
ry");
95:                 }
96:                 cout<< state_dir->get_second(words.at(j))->get_contents()->r
eadfile(); //this works
97:                 cout<< "\n";
98:             }
99:         }
100:     }
101: //go back to this
102: }
```

```
103:
104: void fn_cd (inode_state& state, const wordvec& words) {
105:     DEBUGF ('c', state);
106:     DEBUGF ('c', words);
107:     if(words.size() == 1 or words.at(1) == "/"){    //if no dir specified,
send to root
108:         state.set_cwd(state.get_root());
109:     }
110:     else if(words.size()>2){
111:         throw command_error ("cd: Invalid number of arguments");
112:     }
113:     else{
114:         //if it is a path, then go to last one, checking validity
115:         //to make sure each one is a directory, then set cwd to last dir
116:         shared_ptr <directory> state_dir = dynamic_pointer_cast<directory>
117:             (state.get_cwd()->get_contents());
118:         wordvec split_path = split(words.at(1), "/");
119:         int count = 0;
120:         //if its a path
121:         if(split_path.size()>1){
122:             for(auto i =split_path.begin(); i< split_path.end()-1;i++){ //s
kip last so can set_cwd?
123:                 //if file does not exist at all
124:                 if(state_dir->file_dne(split_path.at(count)) == true){
125:                     throw command_error ("cd: "+words.at(1)+": No such file o
r directory");
126:                 }
127:                 //if exists but is not a directory
128:                 else if(state_dir->is_dir_(split_path.at(count)) == false){
129:                     throw command_error ("cd: "+ words.at(1) +": No such file
or directory");
130:                 }
131:                 count++;
132:             }
133:             if(state_dir->file_dne(split_path.at(count)) == true){
134:                 throw command_error ("cd: "+words.at(1)+": No such file o
r directory");
135:             }
136:             //if exists but is not a directory
137:             else if(state_dir->is_dir_(split_path.at(count)) == false){
138:                 throw command_error ("cd: "+ words.at(1) +": No such file or
directory");
139:             }
140:             else{
141:                 state.set_cwd(state_dir->get_second(words.at(count)));
142:                 //something wrong with setting the cwd?
143:             }
144:         }
145:         else{
146:             if(state_dir->file_dne(words.at(1)) == true){
147:                 throw command_error ("cd: "+words.at(1)+": No such file o
r directory");
148:             }
149:             //if exists but is not a directory
150:             else if(state_dir->is_dir_(words.at(1)) == false){
151:                 throw command_error ("cd: "+ words.at(1) +": No such file or
directory");
152:             }

```

```
153:         else{
154:             state.set_cwd(state_dir->get_second(words.at(1)));
155:         }
156:
157:     }
158: }
159: }
160: //dont have to do anything???
161: void fn_echo (inode_state& state, const wordvec& words) {
162:     DEBUGF ('c', state);
163:     DEBUGF ('c', words);
164:     cout << word_range (words.cbegin() + 1, words.cend()) << endl;
165: }
166:
```

```
167:
168: void fn_exit (inode_state& state, const wordvec& words) {
169:     DEBUGF ('c', state);
170:     DEBUGF ('c', words);
171:     throw ysh_exit();
172: }
173:
174: void fn_ls (inode_state& state, const wordvec& words) {
175:     DEBUGF ('c', state);
176:     DEBUGF ('c', words);
177:     if(words.size()>2){
178:         throw command_error ("ls: Invalid number of arguments"); //dont wo
rk
179:     }
180:     else{
181:         //if one arg, then cwd is used
182:         if(words.size()==1){
183:             // print_ls(state.get_cwd());
184:         }
185:         else{//use arg specified
186:             // print_ls(words.at(1)->get_second());
187:         }
188:     }
189: }
190: }
191:
192: void fn_lsr (inode_state& state, const wordvec& words) {
193:     DEBUGF ('c', state);
194:     DEBUGF ('c', words);
195: }
196:
197: void fn_make (inode_state& state, const wordvec& words) {
198:     DEBUGF ('c', state);
199:     DEBUGF ('c', words);
200:     shared_ptr <directory> state_dir = dynamic_pointer_cast<directory>
201:         (state.get_cwd()->get_contents());
202:     if(words.size()==1){
203:         throw command_error ("make: No target specified"); //dont work
204:     }
205:     wordvec split_path = split(words.at(1), "/"); //skips make call but inc
ludes all paths
206:     int count = 0;
207:     //is a path
208:     if(split_path.size()>1){ //if its a path
209:         for(auto i =split_path.begin(); i< split_path.end()-1;i++){ //skip
last cause creating last so wont exist
210:             //if file does not exist at all
211:             if(state_dir->file_dne(split_path.at(count)) == true){
212:                 throw command_error ("make: "+split_path.at(count)+" : No suc
h file or directory");
213:                 //like foo/wrongdir/bar/newfile
214:                 //would output wrongdir: no such file or dir
215:             }
216:             //if file exists but is not a directory, needs to be while in t
he path
217:             else if(state_dir->is_dir_(split_path.at(count)) == false){
218:                 throw command_error ("make: "+split_path.at(count)+" : No suc
h file or directory");
```

```
219:         }
220:         count++; //cause the auto is an itr
221:     }
222:     //after path, file being made
223:     if(state_dir->file_dne(split_path.at(count))==false){// if file ex
ists
224:         if(state_dir->is_dir_(split_path.at(count)) == true){
225:             throw command_error ("make: "+ words.at(1) +": Is a director
y");
226:         }
227:         else
228:         { //updates file if exists
229:             inode_ptr updated_pointer = state_dir->update_file(split_pat
h.at(count),wordvec(words.begin()+2,words.end()));
230:         }
231:     }
232:     else{//if file doesnt exist already, makes it
233:         inode_ptr new_file = state.get_cwd()->get_contents()->mkfile(sp
lit_path.at(count));
234:         //+2 makes it not include make or filename, jsut contents
235:         new_file->get_contents()->writefile(wordvec(words.begin()+2,wor
ds.end()));
236:     }
237: }
238: //no path, this works!
239: else{
240:     if(state_dir->file_dne(words.at(1))==true){//this works
241:         //if file dne then can do normal cause isnt dir or file
242:         inode_ptr new_file = state.get_cwd()->get_contents()->mkfile(wor
ds.at(1));
243:         //+2 makes it not include make or filename, jsut contents
244:         new_file->get_contents()->writefile(wordvec(words.begin()+2,wor
ds.end()));
245:     }
246:     else{//not dir, but updating existing file
247:         if(state_dir->is_dir_(words.at(1))==true){//this works
248:             throw command_error ("make: "+ words.at(1) +": Is a directo
ry");
249:         }
250:         else{
251:             inode_ptr updated_pointer = state_dir->update_file(words.at(
1),wordvec(words.begin()+2,words.end()));
252:             // cout<< updated_pointer->get_contents()->readfile(); //makes
the file, but includes "make"??
253:         }
254:     }
255: }
256: }
257:
258: void fn_mkdir (inode_state& state, const wordvec& words) {
259:     DEBUGF ('c', state);
260:     DEBUGF ('c', words);
261:     if(words.size()==1){
262:         throw command_error ("mkdir: No target specified"); //dont work
263:     }
264:     else if(words.size()>2){
265:         throw command_error ("mkdir: Invalid number of arguments"); //dont
work
```

```
266:     }
267:
268:     cout<< "in fn_mkdir\n";
269:     wordvec split_path = split(words.at(1), "/"); //skips mkdir call but in
cludes all paths
270:     cout<< "made split path\n";
271:     shared_ptr<directory> state_dir = dynamic_pointer_cast<directory>
272:         (state.get_cwd()->get_contents());
273:     //if has path, check if valid
274:     cout<< "made state_dir\n";
275:     cout<< "split_path.size(): ";
276:     cout<< split_path.size();
277:     int count = 0;
278:     if(split_path.size()>1){ //if its a path
279:         for(auto i =split_path.begin(); i< split_path.end()-1;i++){ //s
kip last cause creating last so wont exist
280:             if(state_dir->file_dne(split_path.at(count))==true){ //this w
orks //everything needs to exist but the last one
281:                 throw command_error ("mkdir: "+split_path.at(count)+" : No
such file or directory"); //dont work
282:                 //print whole path?
283:                 count++; //cause the auto is an itr
284:             }
285:             if(state_dir->file_dne(split_path.at(count)) == false){
286:                 if(state_dir->is_dir_(split_path.at(count)) == true){
287:                     throw command_error ("mkdir: "+split_path.at(count)+" : Fi
le exists"); //dont work
288:                 }
289:             }
290:             else{
291:                 cout<<"setting parent";
292:                 state_dir->mkdir(split_path.at(count));
293:             }
294:         }
295:         //if no path, or end of path, make sure dne
296:         //should be end if has path or not right?
297:         else{//if its not a path
298:             if(state_dir->file_dne(words.at(1)) == false){
299:                 if(state_dir->is_dir_(words.at(1)) == true){
300:                     throw command_error ("mkdir: "+words.at(1)+" : File exists
"); //dont work
301:                 }
302:                 cout<<"setting parent 2";
303:                 // state_dir->get_second(words.at(1))->set_parent(state.get_c
wd());
304:                 state_dir->mkdir(words.at(1));
305:             }
306:             else{
307:                 cout<<"setting parent 2";
308:
309:                 //state_dir->get_second(words.at(1))->set_parent(state.get_cw
d());
310:                 state_dir->mkdir(words.at(1));
311:             }
312:         }
313:     }
314:
315: void fn_prompt (inode_state& state, const wordvec& words) {
```



```
316:     DEBUGF ('c', state);
317:     DEBUGF ('c', words);
318: }
319: //just doesnt print path? segfaults with ivalid paths
320: //this works with the test cases but thats it i think??. idk its good f
or now
321: void fn_pwd (inode_state& state, const wordvec& words) {
322:     DEBUGF ('c', state);
323:     DEBUGF ('c', words);
324:     if(words.size()>2){
325:         throw command_error ("mkdir: Invalid number of arguments"); //dont
work
326:     }
327:     else{
328:         //cout<< "in else\n";
329:         //print from root to cwd
330:         // wordvec path =split(state.path(state.get_cwd()),"/");
331:         // cout<< "\nmade path\n";
332:         /*int i= path.size()-1;
333:         // cout<< "path size:";
334:         // cout<< path.size();
335:         while(i>0){
336:             cout<<"/";
337:             cout<< path.at(i);    //this works, ish
338:             i--;
339:         }*/
340:     }
341:     state.path(state.get_cwd());
342: }
343:
344: void fn_rm (inode_state& state, const wordvec& words) {
345:     DEBUGF ('c', state);
346:     DEBUGF ('c', words);
347: }
348:
349: void fn_rmr (inode_state& state, const wordvec& words) {
350:     DEBUGF ('c', state);
351:     DEBUGF ('c', words);
352: }
353:
```

```
1: // $Id: debug.h,v 1.12 2019-10-16 15:17:26-07 - - $
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 1.15 2020-01-22 14:21:55-08 - - $
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: // 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 << "DEBUG(" << flag << ") "
32:          << file << "[" << line << "]" " << endl
33:          << "... " << pretty_function << endl;
34: }
35:
```

```
1: // $Id: file_sys.h,v 1.10 2021-05-03 17:36:57-07 - - $
2:
3: #ifndef __INODE_H__
4: #define __INODE_H__
5:
6: #include <exception>
7: #include <iostream>
8: #include <memory> //contains shared pointer
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: //implement directory file pointer? maybe not
26: ostream& operator<< (ostream&, file_type);
27:
```

```
28:
29: // inode_state -
30: //   A small convenient class to maintain the state of the simulated
31: //   process: the root (/), the current directory (.), and the
32: //   prompt.
33:
34: class inode_state { //only one can exist in the entire filesystem
35: //inode state does not inherit from inode
36: //however they are cooperating classes
37:
38:     friend class inode;
39:     friend ostream& operator<< (ostream& out, const inode_state&);
40:     //must be a friend not a member
41:     private:
42:         inode_ptr root ;
43:         inode_ptr cwd ;    //need to make a different assignment
44:         //to them in the constructor!!!
45:         string prompt_ {"% "}; //cant have the fuction and field name be
the same
46:     public:
47:         virtual ~inode_state();    //default or make own?
48:         inode_state (const inode_state&) = delete; // copy ctor
49:         inode_state& operator= (const inode_state&) = delete; // op=
50:         //-delete says that if u attempt to copy it the compiler
51:         //will refuse
52:         inode_state(); //constructor, have to do some work
53:         //need to make a destructor?
54:         //need to make a new inode and point root and cwd at it
55:
56:         const string& prompt() const;
57:         //returns prompt when need to print stuff out
58:         void prompt (const string&); //implement later?
59:         //sets the prompts
60:
61:         inode_ptr get_root();
62:         inode_ptr get_cwd();
63:         void set_cwd(inode_ptr);//??? //yes
64:         void path(const inode_ptr& state_cwd);
65:
66:
67: };
68:
69: // class inode -
70: // inode ctor -
71: //   Create a new inode of the given type.
72: // get_inode_nr -
73: //   Retrieves the serial number of the inode. Inode numbers are
74: //   allocated in sequence by small integer.
75: // size -
76: //   Returns the size of an inode. For a directory, this is the
77: //   number of dirents. For a text file, the number of characters
78: //   when printed (the sum of the lengths of each word, plus the
79: //   number of words.(spaces in between words)
80: //
81:
82: class inode {
83:
84:     //inode does not inherit from inode state
```

```
85:     friend class inode_state;
86:     private:
87:         static size_t next_inode_nr; //next inode number
88:         size_t inode_nr; //inode number itself
89:         base_file_ptr contents; //basefile is the abstract base class
90:                                 //directory and plainfile as subclasses
91:                                 //dir and pl do all the work
92:         //inode contains a pointer to base file
93:         bool is_dir; //to know the filetype
94:         inode_ptr parent{nullptr}; //initializes it to null in case there
isnt a parent?
95:     public:
96:         virtual ~inode() = default; //destructor?
97:         inode (file_type); //gets filetype or creates filetype?
98:         size_t get_inode_nr() const; //only copying pointers to inodes
99:         //have to set inode number?
100:        void set_contents(base_file_ptr); //setter
101:        base_file_ptr get_contents(); //getter
102:
103:        bool isdir(); //getter need this??
104:
105:        inode_ptr get_parent(); //need these?
106:        void set_parent(inode_ptr);
107:        string filename = ""; //to keep track of filenames
108:
109: };
110:
```

```
111:
112: // class base_file -
113: // Just a base class at which an inode can point. No data or
114: // functions. Makes the synthesized members useable only from
115: // the derived classes.
116:
117: class file_error: public runtime_error {
118:     public:
119:         explicit file_error (const string& what); //its constructor takes
in a string
120: };
121:
122: class base_file {
123:     protected:
124:         base_file() = default; //basefile has no fields! just an abstract
class
125:         //becasue we have constructors defined, we need to specify defa
ult
126:         virtual const string& error_file_type() const = 0;
127:         //says that this function does not exist in basefile, is an abstra
ct function
128:         //must be overwritten in a subclass
129:     public:
130:         virtual ~base_file() = default;
131:         //if you dont specify a destructor, it will be specified for yo
u^
132:         //must declare destructor as virtual, so it doesnt just delete
133:         //pointers?
134:         base_file (const base_file&) = delete;
135:         //movers, not allowing files to be moved
136:         //means that the implicity generated copier will be prohibited
137:         //dont want to allow base files to be copied, use POINTERS
138:         base_file& operator= (const base_file&) = delete;
139:         virtual size_t size() const = 0;
140:         //base file has no meaning so it is 0
141:         //no meaning because can be one thing on a plain and another on a
directory file
142:         virtual const wordvec& readfile() const;
143:         //will read text from file
144:         virtual void writefile (const wordvec& newdata);
145:         //writes text to file, only available to text files
146:         //error will be thrown for directory file
147:         virtual void remove (const string& filename);
148:         //only appropriate if we are dealing with directories?
149:         virtual inode_ptr mkdir (const string& dirname);
150:         virtual inode_ptr mkfile (const string& filename);
151:         virtual map<string,inode_ptr>& get_dirents();
152:         // virtual bool is_dir();
153: };
```



```
154:
155: // class plain_file -
156: // Used to hold data.
157: // synthesized default ctor -
158: //     Default vector<string> is a an empty vector.
159: // readfile -
160: //     Returns a copy of the contents of the wordvec in the file.
161: // writefile -
162: //     Replaces the contents of a file with new contents.
163:
164: class plain_file: public base_file {
165:     //friend ostream?
166:     private:
167:         wordvec data;
168:         virtual const string& error_file_type() const override {
169:             static const string result = "plain file";
170:             return result; //this is ok? no change?
171:             //if you try and run a function on the wrong type of file,
172:             //returns the filetype
173:             //ex remove f , error f is a plainfile
174:             //static to avoid creating it every time we call a function
175:         }
176:     public:
177:         virtual ~plain_file() = default;
178:         virtual size_t size() const override; //must be overridden
179:         //set size??
180:         virtual const wordvec& readfile() const override; //override erro
r functions in base class
181:         //could make base class functions abstract then require them to be
overwritten
182:         //in base class?
183:         //should use keyword override
184:         //cant override something that doesnt exist in the base class
185:         // would still compile but makes a compile time error so def use o
verride
186:         //to make it easier to debug
187:         //keyword virtual is optional
188:         // virtual void remove (const string& filename)override;
189:         //virtual inode_ptr mkdir (const string& dirname) override; //er
ror will be thrown if basefile version is called
190:                                     //, imple
ment in directory
191:         virtual void writefile (const wordvec& newdata) override;
192:         //virtual inode_ptr mkfile (const string& filename) override;
193:
194:         // virtual bool is_dir() override;
195:
196: };
197:
198: // class directory -
199: // Used to map filenames onto inode pointers.
200: // default ctor -
201: //     Creates a new map with keys "." and "..".
202: // remove -
203: //     Removes the file or subdirectory from the current inode.
204: //     Throws an file_error if this is not a directory, the file
205: //     does not exist, or the subdirectory is not empty.
206: //     Here empty means the only entries are dot (.) and dotdot (..).
```

```
207: // mkdir -
208: //     Creates a new directory under the current directory and
209: //     immediately adds the directories dot (.) and dotdot (..) to it.
210: //     Note that the parent (..) of / is / itself. It is an error
211: //     if the entry already exists.
212: // mkfile -
213: //     Create a new empty text file with the given name. Error if
214: //     a dirent with that name exists.
215:
216: class directory: public base_file {
217:     private:
218:         // Must be a map, not unordered_map, so printing is lexicographic
219:         map<string,inode_ptr> dirents;
220:         virtual const string& error_file_type() const override {
221:             static const string result = "directory";
222:             return result;
223:         }
224:     public:
225:         virtual ~directory() = default;
226:         virtual size_t size() const override; //????? need to override
227:         virtual void remove (const string& filename) override; //need to
implement to throw error?
228:                                     //if direc
tory name is called
229:         virtual inode_ptr mkdir (const string& dirname) override;
230:         virtual inode_ptr mkfile (const string& filename) override;
231:         //virtual void writefile (const wordvec& newdata) override; //????
232:         //make a num files
233:         virtual map<string,inode_ptr>& get_dirents() override;
234:         // virtual bool is_dir() override;
235:         inode_ptr get_second(const string& filename); //need to get rid o
f?
236:         bool file_dne(const string& words);
237:         bool is_dir_(const string& words); //getter need this??
238:         inode_ptr update_file(const string& filename, const wordvec& words)
;
239: };
240:
241: #endif
242:
```

```
1: // $Id: file_sys.cpp,v 1.11 2021-05-03 17:36:57-07 - - $
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}; //initialized to 1, the first inode num
ber
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:
24:
25:
26: inode_state::inode_state() {
27:     // DEBUGF ('i', "root = " << root << ", cwd = " << cwd
28:     //          << ", prompt = \" " << prompt() << "\"");
29:
30:     //inode state constructor
31:     //establish inode state
32:     //create root directory /
33:     //make sure root directory (parent ..) points at itself
34:     //can call inode constructor and pass in a filetype /
35:     //then modify it after the fact?
36:     //can call the make shared plain file and directory/?
37:     //i node and inode state are friends
38:     //which means once you have an inode the inode state can go in and za
P
39:     //the fields in appropriate manners
40:     root = make_shared <inode> (file_type::DIRECTORY_TYPE); //right?
41:     // shared_ptr <directory> root_dir = dynamic_pointer_cast<directory>
42:     //                               (root->get_contents());
43:     //shared_ptr <directory>
44:     cwd = root;
45:     root->parent = root;
46:     pair <string, inode_ptr> dot = {".", root}; //sets dot, cwd, to root
47:     (root->get_contents()->get_dirents()).insert(dot);
48:
49:     pair <string, inode_ptr> dot_dot = {"..", root}; //sets dot dot, th
e parent to root
50:     (root->get_contents()->get_dirents()).insert(dot_dot);
51:
52: }
53: //inode_state method implementations
54: void inode_state::prompt(const string& s){
55:     prompt_ = s;
```

```
56: } //implement later? its ok
57:     //sets the prompts
58: const string& inode_state::prompt() const { return prompt_; }
59: //just returns the prompt
60:
61:
62: inode_ptr inode_state::get_root(){ return root; }
63:
64: inode_ptr inode_state::get_cwd(){ return cwd; } //need this?
65:
66: void inode_state::set_cwd(inode_ptr new_cwd){
67:     cwd = new_cwd;
68:     cout<<"new cwd:";
69:     cout<< cwd->filename;
70: }
71: void inode_state::path(const inode_ptr& state_cwd){
72:
73:     wordvec rootpath;
74:     cout<<"made rootpath\n";
75:     rootpath.push_back("/");
76:     rootpath.push_back(state_cwd->filename); //then pushback other names?
77:     cout<< "pushed back first time\n filename:";
78:     cout<<"rootpath:";
79:     cout<< rootpath;
80:
81:     inode_ptr cwd_dirents = state_cwd->get_contents()->get_dirents().find
("..")->second;
82:     cout<<"\nset cwd_dirents";
83:     while(cwd_dirents->inode_nr!= 1){
84:         cout<< "\n inwhile";
85:         cwd_dirents = cwd_dirents->get_contents()->get_dirents().find("..
")->second; //issue here!->use dot dot not parent!!
86:         cout<< "in while cwd_dirents changed\n";
87:         //cout<<"/";
88:         // cout<<cwd_dirents->filename;
89:
90:         rootpath.push_back(cwd_dirents->filename);
91:         rootpath.push_back("/");
92:         cout<< "in while\n rootpath pushedback:";
93:         cout<<rootpath;
94:
95:     }
96:     cout<<"after while";
97:     string path_output;
98:     for (unsigned long i = 0; i>rootpath.size();i++){
99:         //path_output.append(rootpath.at(i));
100:         cout<< rootpath.at(i);
101:     }
102:     // cout<< path_output;
103:
104: }
105:
106:
107:
108: void rm_r( inode_ptr roo){
109:     //depth first search (postorder)
110:     map<string,inode_ptr>& roo_dirents = (roo->get_contents()->get_dirent
s());
```

```
111: //create map of dirents of the file roo
112: for(auto ritor = roo_dirents.crbegin(); ritor != roo_dirents.crend();
++ritor){ //cr or nah
113:     //recur over each entry other than dot or dot dot
114:     if(ritor->first!="." and ritor->first != ".."
115:         and ritor->second->isdir()
116:         ==true){//->get_contents()?
117:         rm_r(ritor->second);
118:     }
119:     //if not directory, or empty directory, erase
120:     roo_dirents.erase(ritor->first);
121: }
122: roo_dirents.erase("."); //erasing root last
123: roo_dirents.erase(".."); //erasing root last
124:
125: }
126:
127:
128: //////////// inode_state destructor//////////
129:
130: inode_state::~inode_state(){
131:
132:     rm_r(root);
133:     cwd = nullptr;//need to do this?
134:     root = nullptr;//idk
135: }
136:
137:
138: ostream& operator<< (ostream& out, const inode_state& state) {
139:     //just prints out inode state
140:     //just used in debug statements in working code
141:     out << "inode_state: root = " << state.root//machine addresses
142:         << ", cwd = " << state.cwd;
143:     return out;
144: }
145: //////////////////////////////////////////inode////////////////////////////////////
146: inode::inode(file_type type): inode_nr (next_inode_nr++) {
147:     //constructor
148:     //need a virtual constructor but no such thing in c++
149:     //so instead pass in an argument
150:     //default constructor on an inode has been suppressed
151:     //so just say new node and give it the particular filetype that you want to
152:     //create
153:     //depends on the command it is being called from
154:     file_type = type;
155:     switch (type) {
156:         case file_type::PLAIN_TYPE:
157:             is_dir = false;
158:             contents = make_shared<plain_file>();
159:             break;
160:         case file_type::DIRECTORY_TYPE:
161:             is_dir = true;
162:             contents = make_shared<directory>(); //make shared of a plain
or directory
163:             //adjust the file sysem
164:             //making filesystem friends
165:             //inode and inode state are already friends
```

```
166:             //base file probably doesnt work well without inode
167:             //information hiding is not important?? idk
168:             break;
169:             default: assert (false);    //for the sake of clarity
170:         }
171:         //F ('i', "inode " << inode_nr << ", type = " << type);
172:     }
173:
174: size_t inode::get_inode_nr() const {
175:
176:     //just gets inode number, already done
177:     // DEBUGF ('i', "inode = " << inode_nr);
178:     return inode_nr;
179: }
180: //void inode::set_contents(base_file_ptr new_contents){
181: //    contents = new_contents;
182: //}    //dont ever need to set new contents though right?
183: base_file_ptr inode::get_contents(){ return contents; } //getter
184:
185: bool inode::isdir(){
186:     return is_dir; } //getter need this??
187: //or just use is_dir
188:
189: inode_ptr inode::get_parent(){
190:     cout<<"get_parent filename:\n";
191:     cout<<parent->filename;
192:     return parent;
193: }
194: void inode::set_parent(inode_ptr dir){
195:
196:     parent = dir;
197:     cout<<"get_parent filename:";
198:     cout<<parent->filename;
199: }
200:
201:
202:
203: file_error::file_error (const string& what):
204: //implementation of a file error could have been done in line? idk
205: //need to change?
206:     runtime_error (what) {
207: }
208:
209: //all these functions do is throw a file error based on the file type
210: //that is in basefile
211: //those fucntions will either be inherited or overwritten
212: //if dont override, will be inherited
213:
214: //can leave alone until plainfile size
215: const wordvec& base_file::readfile() const {
216:
217:     throw file_error ("readfile: is a " + error_file_type());
218: }
219:
220: void base_file::writefile (const wordvec&) {
221:     throw file_error ("writefile: is a " + error_file_type());
222: }
223:
```

```
224: void base_file::remove (const string&) {
225:     throw file_error ("remove: is a " + error_file_type());
226: }
227:
228: inode_ptr base_file::mkdir (const string&) {
229:     throw file_error ("mkdir:is a " + error_file_type());
230: }
231:
232: inode_ptr base_file::mkfile (const string&) {
233:     throw file_error ("mkfile:is a " + error_file_type()); //dont work
234: }
235: //added functions
236: map<string,inode_ptr>& base_file::get_dirents() {
237:     throw file_error ("getdirents: is a " + error_file_type()); //dont wo
rk
238: }
239: /*bool base_file::is_dir() {
240:     throw file_error ("isdir: is a " + error_file_type()); //dont work
241: }*/
242:
```

```
//plainfile must override read and writefile
244: //but can go ahead and inherit remove mkdir and mkfile
245: //all of these need to be done!!!
246: size_t plain_file::size() const { //constant function
247:     //use wordvec data
248:     size_t size = data.size(); //does this work
249:     // DEBUGF ('i', "size = " << size);{
250:     for(auto i = data.begin();i<data.end();i++){ //will there be an extra
space?
251:         size += i->size();//each word
252:     }
253:     return data.size(); //calling size function from map?
254: }
255:
256: const wordvec& plain_file::readfile() const {
257:     // DEBUGF ('i', data);
258:     return data; //dont change?
259: }
260:
261: void plain_file::writefile (const wordvec& words) {
262:     // DEBUGF ('i', words);//must change
263:     data = words; //sets data to the wordvec words
264: }
265: /*
266: bool plain_file::is_dir() {
267:     return false;
268: }*/
269: //directory must override remove mkdir and mkfile but can inherit
270: //readfile and writefile
271: //all of these need to be done!!!
272:
273: //could just handle plain files initially
274: //because need to make an inode for the root directory
275: //needs to have a directory file in it
276: //but the last three wont be used if dont test using those, dont make or
delete files
277: size_t directory::size() const {
278:     //size_t size = dirents.size(); //can use directory.size function in
map?
279:     // DEBUGF ('i', "size = " << size);
280:     return dirents.size();
281: }
282: //just override the base files
283:
284: void directory::remove (const string& filename) {
285:     //DEBUGF ('i', filename); //needs to delete something from a director
y
286:     //idk look at this more
287:     //if empty directory or if file
288:     //use find() function
289:     //shouldnt work on root though? idk
290:     inode_ptr rm_ptr = dirents.find(filename)->second;
291:     if(rm_ptr->isdir() == false
292:         ||dirents.find(filename)->first != ".."){
293:         dirents.erase(filename);
294:     }
295: }
296:
```



```
297: inode_ptr directory::mkdir (const string& dirname) {
298:     // DEBUGF ('i', dirname); //creates directory
299:     //error if file or directory of same name is already
300:     //created, or if the complete pathname to the parent of
301:     //this dir does not already exist
302:     //dot and dot dot added to dirents
303:
304:     /*if(dirents.find(dirname)->second != inode_ptr()){ //if it has been
created
305:         throw file_error ("mkdir: file already exists: " + dirname); //thr
ow error
306:     }*/
307:     cout<< "in mkdir";
308:     inode_ptr newDir = make_shared<inode>(file_type::DIRECTORY_TYPE);
309:     //make new dir
310:     newDir->filename = dirname;
311:     //newDir->set_parent(this.get_cwd());
312:     cout<<"\nnewDir filename:";
313:     cout<< newDir->filename;
314:     //cout<<"\nparent filename:";
315:     //cout<< newDir->get_parent()->filename; //parents arent changing
316:     //insert new dir to dirents
317:     cout<<"\ninserting new pair\n";
318:     pair<string, inode_ptr> newPair = {dirname, newDir};
319:     dirents.insert(newPair);
320:     cout<<"inserted new pair\n";
321:     // newDir->set_parent(newDir->get_cwd());
322:     //add dot/dotdot to current dir
323:     pair<string, inode_ptr> dot = {".", newDir}; //sets dot, cwd
324:     //(newDir->get_contents()->get_dirents()).insert(dot);
325:     cout<<"made dot \n";
326:     dirents.insert(dot);
327:     cout<<" inserted dot\n";
328:     pair<string, inode_ptr> dot_dot = {"..", dirents.find(".")->second};
; //sets dot dot, the paren(cwd before new dir)
329:     //(newDir->get_contents()->get_dirents()).insert(dot_dot);
330:     cout<<" made dot dot\n";
331:     dirents.insert(dot_dot);
332:     cout<<" inserted dot dot\n";
333:     newDir->set_parent(dirents.find(".")->second);
334:     cout<< newDir->get_parent();
335:     // cout<<newDir->get_contents()->get_dirents().find("..")->second->fi
lename;
336:     return newDir;
337: }
338:
339: inode_ptr directory::mkfile (const string& filename) {
340:     //DEBUGF ('i', filename); //creates file
341:     //file specified is created and the rest of the wordst
342:     //are put in that file
343:     //if the file already exists, a new one is not created but the
344:     //contents are replaced
345:     //error to specify a directory
346:     //if there are no words the file is empty
347:     //inode_ptr i_node_ptr = dirents.find(filename)->second;
348:     //if(i_node_ptr->isdir() == true){
349:         throw file_error ("mkfile: file is a directory " + filename); /
/throw error
```

```
350:     */
351:     //make new file
352:     inode_ptr newFile = make_shared<inode>(file_type::PLAIN_TYPE);
353:     newFile->filename = filename;
354:     // cout<< "\nfilename:";
355:     // cout<< filename;
356:     //newFile->set_parent(inode_state().get_cwd());
357:     //insert/replace contents
358:     pair<string,inode_ptr> newFilePair = {filename,newFile};
359:     dirents.insert(newFilePair); //dirents[filename]= newFile;
360:     return newFile;
361: }
362:
363: map<string,inode_ptr>& directory::get_dirents() {
364:     return dirents;
365: }
366: /*bool directory::is_dir() {
367:     return true;
368: }*/
369: /*inode_ptr directory::get_cwd(){
370:     dirents.find(".")->second;
371: }
372: */
373: //void directory::writefile (const wordvec&) {
374: //    throw file_error ("writefile: is a " + error_file_type());
375: //}
376: bool directory::file_dne( const string& str){
377:     if(dirents.find(str)== dirents.end()){
378:         return true;
379:     }
380:     return false;
381: }
382: bool directory::is_dir_(const string& words){
383:     return( dirents.find(words)->second->isdir());
384: }
385:
386: inode_ptr directory::update_file(const string& filename, const wordvec&
words){
387:     inode_ptr update_ptr = dirents.find(filename)->second;
388:     update_ptr->get_contents()->writefile(words);
389:     pair<string,inode_ptr> update_pair = {filename,update_ptr};
390:     dirents.insert(update_pair); //dirents[filename]= newFile;
391:     return update_ptr;
392: }
393: inode_ptr directory::get_second(const string& filename){
394:     cout<<"checking for seg fault";
395:     return dirents.find(filename)->second;
396: }
397:
398:
399:
400:
401:
402:
```

```
1: // $Id: util.h,v 1.14 2020-10-22 18:00:02-07 - - $
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 1.14 2019-10-08 14:01:38-07 - - $
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 2021-05-03 17:36:57-07 - - $
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:                 //if(words.size()>1){
72:                 string word0 = words.at(0);
73:                 if(!(word0.at(0)=='#')){
74:                     command_fn fn = find_command_fn (words.at(0));
75:                     fn (state, words);
76:                 }
77:                 // }
78:             } catch (file_error& error) {
79:                 complain() << error.what() << endl;
80:             } catch (command_error& error) {
81:                 complain() << error.what() << endl;
82:             }
83:         }
84:     } catch (ysh_exit&) {
85:         // This catch intentionally left blank.
86:     }
87:
88:     return exit_status_message();
89: }
90:
```



```
1: # $Id: Makefile,v 1.42 2021-05-03 17:36:57-07 - - $
2:
3: MKFILE      = Makefile
4: DEPPFILE    = ${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++17 -g -O0 ${GPPOPTS}
11: MAKEDEPCPP  = g++ -std=gnu++17 -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} ${DEPPFILE}
44:
45: clean :
46:         - rm ${OBJECTS} ${DEPPFILE} 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:     ${GMAKE} spotless dep ci all lis
62: submit:
63:     submit cse111-wm.s21 asg2 *.cpp *.h Makefile README
64:
65: ifeq (${NEEDINCL}, )
66: include ${DEPFILE}
67: endif
68:
```

```
1: # Makefile.dep created Mon May  3 17:36:56 PDT 2021
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
```