

```
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 2021-04-30 15:22:12-07 - - $
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:     }
52:     else{
53:         for(unsigned long i = 1;i < words.size(); ++i){
54:             shared_ptr <directory> state_dir = dynamic_pointer_cast<directory>
55:                 (state.get_cwd()->get_contents());
56:             //accounts for more than one file, if file is a directory
```

```
56:         if(state.get_cwd()->get_file_type() == file_type::DIRECTORY_TYPE
){
57:             throw file_error ("cat: "+ words.at(i) +": is a directory");
58:         }
59:         //if file does not exist, find returns the last
60:         else if(state_dir->get_dirents().find(words.at(i))->first == ""
){
61:             throw file_error ("cat: "+ words.at(i) +": No such file or
directory");
62:         }
63:
64:         cout<< state_dir->get_dirents().find(words.at(i))->first;
65:     }
66: }
67:
68: }
69:
70: void fn_cd (inode_state& state, const wordvec& words) {
71:     DEBUGF ('c', state);
72:     DEBUGF ('c', words);
73: }
74:
75: void fn_echo (inode_state& state, const wordvec& words) {
76:     DEBUGF ('c', state);
77:     DEBUGF ('c', words);
78:     cout << word_range (words.cbegin() + 1, words.cend()) << endl;
79: }
80:
```

```
81:
82: void fn_exit (inode_state& state, const wordvec& words) {
83:     DEBUGF ('c', state);
84:     DEBUGF ('c', words);
85:     throw ysh_exit();
86: }
87:
88: void fn_ls (inode_state& state, const wordvec& words) {
89:     DEBUGF ('c', state);
90:     DEBUGF ('c', words);
91: }
92:
93: void fn_lsr (inode_state& state, const wordvec& words) {
94:     DEBUGF ('c', state);
95:     DEBUGF ('c', words);
96: }
97:
98: void fn_make (inode_state& state, const wordvec& words) {
99:     DEBUGF ('c', state);
100:    DEBUGF ('c', words);
101: }
102:
103: void fn_mkdir (inode_state& state, const wordvec& words) {
104:     DEBUGF ('c', state);
105:     DEBUGF ('c', words);
106: }
107:
108: void fn_prompt (inode_state& state, const wordvec& words) {
109:     DEBUGF ('c', state);
110:     DEBUGF ('c', words);
111: }
112:
113: void fn_pwd (inode_state& state, const wordvec& words) {
114:     DEBUGF ('c', state);
115:     DEBUGF ('c', words);
116: }
117:
118: void fn_rm (inode_state& state, const wordvec& words) {
119:     DEBUGF ('c', state);
120:     DEBUGF ('c', words);
121: }
122:
123: void fn_rmr (inode_state& state, const wordvec& words) {
124:     DEBUGF ('c', state);
125:     DEBUGF ('c', words);
126: }
127:
```

```
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 2021-04-30 15:22:12-07 - - $
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 {nullptr};
43:     inode_ptr cwd {nullptr}; //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:
65: };
66:
67: // class inode -
68: // inode ctor -
69: //   Create a new inode of the given type.
70: // get_inode_nr -
71: //   Retrieves the serial number of the inode. Inode numbers are
72: //   allocated in sequence by small integer.
73: // size -
74: //   Returns the size of an inode. For a directory, this is the
75: //   number of dirents. For a text file, the number of characters
76: //   when printed (the sum of the lengths of each word, plus the
77: //   number of words.(spaces in between words)
78: //
79:
80: class inode {
81:
82:     //inode does not inherit from inode state
83:     friend class inode_state;
84: private:
```

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

```
107:
108: // class base_file -
109: // Just a base class at which an inode can point. No data or
110: // functions. Makes the synthesized members useable only from
111: // the derived classes.
112:
113: class file_error: public runtime_error {
114:     public:
115:         explicit file_error (const string& what); //its constructor takes
in a string
116: };
117:
118: class base_file {
119:     protected:
120:         base_file() = default; //basefile has no fields! just an abstract
class
121:         //becasue we have constructors defined, we need to specify defa
ult
122:         virtual const string& error_file_type() const = 0;
123:         //says that this function does not exist in basefile, is an abstra
ct function
124:         //must be overwritten in a subclass
125:     public:
126:         virtual ~base_file() = default;
127:         //if you dont specify a destructor, it will be specified for yo
u^
128:         //must declare destructor as virtual, so it doesnt just delete
129:         //pointers?
130:         base_file (const base_file&) = delete;
131:         //movers, not allowing files to be moved
132:         //means that the implicity generated copier will be prohibited
133:         //dont want to allow base files to be copied, use POINTERS
134:         base_file& operator= (const base_file&) = delete;
135:         virtual size_t size() const = 0;
136:         //base file has no meaning so it is 0
137:         //no meaning because can be one thing on a plain and another on a
directory file
138:         virtual const wordvec& readfile() const;
139:         //will read text from file
140:         virtual void writefile (const wordvec& newdata);
141:         //writes text to file, only available to text files
142:         //error will be thrown for directory file
143:         virtual void remove (const string& filename);
144:         //only appropriate if we are dealing with directories?
145:         virtual inode_ptr mkdir (const string& dirname);
146:         virtual inode_ptr mkfile (const string& filename);
147:         virtual map<string,inode_ptr>& get_dirents();
148:         // virtual bool is_dir();
149: };
```

```
150:
151: // class plain_file -
152: // Used to hold data.
153: // synthesized default ctor -
154: //     Default vector<string> is a an empty vector.
155: // readfile -
156: //     Returns a copy of the contents of the wordvec in the file.
157: // writefile -
158: //     Replaces the contents of a file with new contents.
159:
160: class plain_file: public base_file {
161:     //friend ostream?
162:     private:
163:         wordvec data;
164:         virtual const string& error_file_type() const override {
165:             static const string result = "plain file";
166:             return result;    //this is ok? no change?
167:             //if you try and run a function on the wrong type of file,
168:             //returns the filetype
169:             //ex remove f , error f is a plainfile
170:             //static to avoid creating it every time we call a function
171:         }
172:     public:
173:         virtual ~plain_file() = default;
174:         virtual size_t size() const override;    //must be overridden
175:         //set size??
176:         virtual const wordvec& readfile() const override;    //override erro
r functions in base class
177:         //could make base class functions abstract then require them to be
overwritten
178:         //in base class?
179:         //should use keyword override
180:         //cant override something that doesnt exist in the base class
181:         // would still compile but makes a compile time error so def use o
verride
182:         //to make it easier to debug
183:         //keyword virtual is optional
184:         // virtual void remove (const string& filename)override;
185:         //virtual inode_ptr mkdir (const string& dirname) override;    //er
ror will be thrown if basefile version is called
186:                                     //, imple
ment in directory
187:         virtual void writefile (const wordvec& newdata) override;
188:         //virtual inode_ptr mkfile (const string& filename) override;
189:
190:         // virtual bool is_dir() override;
191:
192: };
193:
194: // class directory -
195: // Used to map filenames onto inode pointers.
196: // default ctor -
197: //     Creates a new map with keys "." and "..".
198: // remove -
199: //     Removes the file or subdirectory from the current inode.
200: //     Throws an file_error if this is not a directory, the file
201: //     does not exist, or the subdirectory is not empty.
202: //     Here empty means the only entries are dot (.) and dotdot (..).
```

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

```
1: // $Id: file_sys.cpp,v 1.11 2021-04-30 15:22:12-07 - - $
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:
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: }
69: void rm_r( inode_ptr roo){
70:     //depth first search (postorder)
71:     map<string,inode_ptr>& roo_dirents = (roo->get_contents()->get_dirent
s());
72:     //create map of dirents of the file roo
73:     for(auto ritor = roo_dirents.crbegin(); ritor != roo_dirents.crend();
++ritor){ //cr or nah
74:         //recur over each entry other than dot or dot dot
75:         if(ritor->first!="." and ritor->first != ".."
and ritor->second->get_file_type()
76:             ==file_type::DIRECTORY_TYPE){//->get_contents()?
77:             rm_r(ritor->second);
78:         }
79:         //if not directory, or empty directory, erase
80:         roo_dirents.erase(ritor->first);
81:     }
82:     roo_dirents.erase("."); //erasing root last
83:     roo_dirents.erase(".."); //erasing root last
84:
85:
86: }
87: //////////// inode_state destructor//////////
88:
89: inode_state::~inode_state(){
90:
91:     rm_r(root);
92:     cwd = nullptr; //need to do this?
93:     root = nullptr; //idk
94: }
95:
96:
97: ostream& operator<< (ostream& out, const inode_state& state) {
98:     //just prints out inode state
99:     //just used in debug statements in working code
100:     out << "inode_state: root = " << state.root//machine addresses
101:         << ", cwd = " << state.cwd;
102:     return out;
103: }
104: //////////////////////////////////////////inode////////////////////////////////
105: inode::inode(file_type type): inode_nr (next_inode_nr++) {
106:     //constructor
107:     //need a virtual constructor but no such thing in c++
108:     //so instead pass in an argument
109:     //default constructor on an inode has been supressed
110:     //so just say new node and give it the particular filetype that you w
ant to
```

```
111: //create
112: //depends on the command it is being called from
113: fileType = type;
114: switch (type) {
115:     case file_type::PLAIN_TYPE:
116:         contents = make_shared<plain_file>();
117:         break;
118:     case file_type::DIRECTORY_TYPE:
119:         contents = make_shared<directory>(); //make shared of a plain
or directory
120:         //adjust the file sysem
121:         //making filesystem friends
122:         //inode and inode state are already friends
123:         //base file probably doesnt work well without inode
124:         //information hiding is not important?? idk
125:         break;
126:     default: assert (false); //for the sake of clarity
127: }
128: //F ('i', "inode " << inode_nr << ", type = " << type);
129: }
130:
131: size_t inode::get_inode_nr() const {
132:
133:     //just gets inode number, already done
134:     // DEBUGF ('i', "inode = " << inode_nr);
135:     return inode_nr;
136: }
137: //void inode::set_contents(base_file_ptr new_contents){
138: //    contents = new_contents;
139: //} //dont ever need to set new contents though right?
140: base_file_ptr inode::get_contents(){ return contents; } //getter
141:
142: file_type inode::get_file_type(){ return fileType; } //getter need this?
?
143: //or just use is_dir
144:
145: inode_ptr inode::get_parent(){
146:     return parent;
147: }
148:
149:
150:
151:
152: file_error::file_error (const string& what):
153: //implementation of a file error could have been done in line? idk
154: //need to change?
155:     runtime_error (what) {
156: }
157:
158: //all these functions do is throw a file error based on the file type
159: //that is in basefile
160: //those fucntions will either be inherited or overwritten
161: //if dont override, will be inherited
162:
163: //can leave alone until plainfile size
164: const wordvec& base_file::readfile() const {
165:
166:     throw file_error ("readfile: is a " + error_file_type());
```



```
167: }
168:
169: void base_file::writefile (const wordvec&) {
170:     throw file_error ("writefile: is a " + error_file_type());
171: }
172:
173: void base_file::remove (const string&) {
174:     throw file_error ("remove: is a " + error_file_type());
175: }
176:
177: inode_ptr base_file::mkdir (const string&) {
178:     throw file_error ("mkdir:is a " + error_file_type());
179: }
180:
181: inode_ptr base_file::mkfile (const string&) {
182:     throw file_error ("mkfile:is a " + error_file_type()); //dont work
183: }
184: //added functions
185: map<string,inode_ptr>& base_file::get_dirents() {
186:     throw file_error ("getdirents: is a " + error_file_type()); //dont wo
rk
187: }
188: /*bool base_file::is_dir() {
189:     throw file_error ("isdir: is a " + error_file_type()); //dont work
190: }*/
191:
```

```
// Plainfile must override read and writefile
193: //but can go ahead and inherit remove mkdir and mkfile
194: //all of these need to be done!!!
195: size_t plain_file::size() const { //constant function
196:     //use wordvec data
197:     //size_t size = data.size(); //does this work
198:     // DEBUGF ('i', "size = " << size);
199:     return data.size(); //calling size function from map?
200: }
201:
202: const wordvec& plain_file::readfile() const {
203:     // DEBUGF ('i', data);
204:     return data; //dont change?
205: }
206:
207: void plain_file::writefile (const wordvec& words) {
208:     // DEBUGF ('i', words); //must change
209:     data = words; //sets data to the wordvec words
210: }
211: /*
212: bool plain_file::is_dir() {
213:     return false;
214: }*/
215: //directory must override remove mkdir and mkfile but can inherit
216: //readfile and writefile
217: //all of these need to be done!!!
218:
219: //could just handle plain files initially
220: //because need to make an inode for the root directory
221: //needs to have a directory file in it
222: //but the last three wont be used if dont test using those, dont make or
delete files
223: size_t directory::size() const {
224:     //size_t size = dirents.size(); //can use directory.size function in
map?
225:     // DEBUGF ('i', "size = " << size);
226:     return dirents.size();
227: }
228: //just override the base files
229:
230: void directory::remove (const string& filename) {
231:     //DEBUGF ('i', filename); //needs to delete something from a director
Y
232:     //idk look at this more
233:     //if empty directory or if file
234:     //use find() function
235:     //shouldnt work on root though? idk
236:     inode_ptr rm_ptr = dirents.find(filename)->second;
237:     if(rm_ptr->get_file_type() == file_type::PLAIN_TYPE
238:        || dirents.find(filename)->first != ".."){
239:         dirents.erase(filename);
240:     }
241: }
242:
243: inode_ptr directory::mkdir (const string& dirname) {
244:     // DEBUGF ('i', dirname); //creates directory
245:     //error if file or directory of same name is already
246:     //created, or if the complete pathname to the parent of
```

```
247: //this dir does not already exist
248: //dot and dot dot added to dirents
249:
250: if(dirents.find(dirname)->second == inode_ptr()){ //if it has been c
reated
251:     throw file_error ("mkdir: file already exists: " + dirname); //thr
ow error
252: }
253:
254: inode_ptr newDir = make_shared<inode>(file_type::DIRECTORY_TYPE);
255: //make new dir
256:
257: //insert new dir to dirents
258: pair<string,inode_ptr> newPair = {dirname,newDir};
259: dirents.insert(newPair);
260:
261: //add dot/dotdot to current dir
262: pair <string, inode_ptr> dot = {".", newDir}; //sets dot, cwd
263: (newDir->get_contents()->get_dirents()).insert(dot);
264:
265: pair <string, inode_ptr> dot_dot = {"..", inode_state().get_cwd()};
//sets dot dot, the paren(cwd before new dir)
266: (newDir->get_contents()->get_dirents()).insert(dot_dot);
267:
268: return newDir;
269: }
270:
271: inode_ptr directory::mkfile (const string& filename) {
272:     //DEBUGF ('i', filename); //creates file
273: //file specified is created and the rest of the wordst
274: //are put in that file
275: //if the file already exists, a new one is not created but the
276: //contents are replaced
277: //error to specify a directory
278: //if there are no words the file is empty
279:     inode_ptr i_node_ptr = dirents.find(filename)->second;
280:     if(i_node_ptr->get_file_type() == file_type::DIRECTORY_TYPE){
281:         throw file_error ("mkfile: file is a directory " + filename); /
/throw error
282:     }
283:     //make new file
284:     inode_ptr newFile = make_shared<inode>(file_type::PLAIN_TYPE);
285:     //insert/replace contents
286:     pair<string,inode_ptr> newFilePair = {filename,newFile};
287:     dirents.insert(newFilePair);
288:     return newFile;
289: }
290:
291: map<string,inode_ptr>& directory::get_dirents() {
292:     return dirents;
293: }
294: /*bool directory::is_dir() {
295:     return true;
296: }*/
297: /*inode_ptr directory::get_cwd(){
298:     dirents.find(".")->second;
299: }
300: */
```

```
301: //void directory::writefile (const wordvec&) {
302: //    throw file_error ("writefile: is a " + error_file_type());
303: //}
304:
305:
306:
307:
308:
309:
```

```
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 1.11 2020-10-28 11:37:57-07 - - $
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:                 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 2021-04-29 21:33:33-07 - - $
2:
3: MKFILE      = Makefile
4: DEFILE      = ${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} ${DEFILE}
44:
45: clean :
46:     - rm ${OBJECTS} ${DEFILE} 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 ${ALLSOURCES} README
64:
65: ifeq (${NEEDINCL}, )
66: include ${DEPFILE}
67: endif
68:
```

```
1: # Makefile.dep created Fri Apr 30 15:22:11 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
```