

```
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 1.21 2021-04-30 21:39:38-07 - - $
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:         cerr<< "before for \n";
54:         for(unsigned long i = 1;i < words.size(); ++i){
55:             shared_ptr <directory> state_dir = dynamic_pointer_cast<directo
56:             (state.get_cwd()->get_contents());
57:         }
58:     }
59: }
```

```
56:         if(state_dir->file_dne(words.at(i))==true){
57:             throw command_error("cat: "+ words.at(i) +": No such file or
directory");
58:         }
59:         if(state_dir->get_file_helper(words.at(i))==file_type::DIRECTOR
Y_TYPE){//everything is directory?
60:             throw command_error("cat: "+ words.at(i) +": is a directory!
"); //throw command error not cerr or cout
61:         }
62:         //cout<< state_dir->second->get_contents()->readfile();
63:         //.find(words.at(i))->first;*/
64:     }
65: }
66:
67: }
68:
69: void fn_cd (inode_state& state, const wordvec& words) {
70:     DEBUGF ('c', state);
71:     DEBUGF ('c', words);
72: }
73:
74: void fn_echo (inode_state& state, const wordvec& words) {
75:     DEBUGF ('c', state);
76:     DEBUGF ('c', words);
77:     cout << word_range (words.cbegin() + 1, words.cend()) << endl;
78: }
79:
```

```
80:
81: void fn_exit (inode_state& state, const wordvec& words) {
82:     DEBUGF ('c', state);
83:     DEBUGF ('c', words);
84:     throw ysh_exit();
85: }
86:
87: void fn_ls (inode_state& state, const wordvec& words) {
88:     DEBUGF ('c', state);
89:     DEBUGF ('c', words);
90: }
91:
92: void fn_lsr (inode_state& state, const wordvec& words) {
93:     DEBUGF ('c', state);
94:     DEBUGF ('c', words);
95: }
96:
97: void fn_make (inode_state& state, const wordvec& words) {
98:     DEBUGF ('c', state);
99:     DEBUGF ('c', words);
100: }
101:
102: void fn_mkdir (inode_state& state, const wordvec& words) {
103:     DEBUGF ('c', state);
104:     DEBUGF ('c', words);
105: }
106:
107: void fn_prompt (inode_state& state, const wordvec& words) {
108:     DEBUGF ('c', state);
109:     DEBUGF ('c', words);
110: }
111:
112: void fn_pwd (inode_state& state, const wordvec& words) {
113:     DEBUGF ('c', state);
114:     DEBUGF ('c', words);
115: }
116:
117: void fn_rm (inode_state& state, const wordvec& words) {
118:     DEBUGF ('c', state);
119:     DEBUGF ('c', words);
120: }
121:
122: void fn_rmr (inode_state& state, const wordvec& words) {
123:     DEBUGF ('c', state);
124:     DEBUGF ('c', words);
125: }
126:
```

```
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 1.9 2021-04-30 21:39:38-07 - - $
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:
68: // class inode -
69: // inode ctor -
70: //   Create a new inode of the given type.
71: // get_inode_nr -
72: //   Retrieves the serial number of the inode. Inode numbers are
73: //   allocated in sequence by small integer.
74: // size -
75: //   Returns the size of an inode. For a directory, this is the
76: //   number of dirents. For a text file, the number of characters
77: //   when printed (the sum of the lengths of each word, plus the
78: //   number of words.(spaces in between words)
79: //
80:
81: class inode {
82:
83:     //inode does not inherit from inode state
84:     friend class inode_state;
```

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

```
109:
110: // class base_file -
111: // Just a base class at which an inode can point. No data or
112: // functions. Makes the synthesized members useable only from
113: // the derived classes.
114:
115: class file_error: public runtime_error {
116:     public:
117:         explicit file_error (const string& what); //its constructor takes
in a string
118: };
119:
120: class base_file {
121:     protected:
122:         base_file() = default; //basefile has no fields! just an abstract
class
123:         //becasue we have constructors defined, we need to specify defa
ult
124:         virtual const string& error_file_type() const = 0;
125:         //says that this function does not exist in basefile, is an abstra
ct function
126:         //must be overwritten in a subclass
127:     public:
128:         virtual ~base_file() = default;
129:         //if you dont specify a destructor, it will be specified for yo
u^
130:         //must declare destructor as virtual, so it doesnt just delete
131:         //pointers?
132:         base_file (const base_file&) = delete;
133:         //movers, not allowing files to be moved
134:         //means that the implicity generated copier will be prohibited
135:         //dont want to allow base files to be copied, use POINTERS
136:         base_file& operator= (const base_file&) = delete;
137:         virtual size_t size() const = 0;
138:         //base file has no meaning so it is 0
139:         //no meaning because can be one thing on a plain and another on a
directory file
140:         virtual const wordvec& readfile() const;
141:         //will read text from file
142:         virtual void writefile (const wordvec& newdata);
143:         //writes text to file, only available to text files
144:         //error will be thrown for directory file
145:         virtual void remove (const string& filename);
146:         //only appropriate if we are dealing with directories?
147:         virtual inode_ptr mkdir (const string& dirname);
148:         virtual inode_ptr mkfile (const string& filename);
149:         virtual map<string,inode_ptr>& get_dirents();
150:         // virtual bool is_dir();
151: };
```

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

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

```
1: // $Id: file_sys.cpp,v 1.10 2021-04-30 21:39:38-07 - - $
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:
70:
71:
72: void rm_r( inode_ptr roo){
73:     //depth first search (postorder)
74:     map<string,inode_ptr>& roo_dirents = (roo->get_contents()->get_dirent
s());
75:     //create map of dirents of the file roo
76:     for(auto ritor = roo_dirents.crbegin(); ritor != roo_dirents.crend();
++ritor){ //cr or nah
77:         //recur over each entry other than dot or dot dot
78:         if(ritor->first!="." and ritor->first != ".."
79:             and ritor->second->get_file_type()
80:             ==file_type::DIRECTORY_TYPE){//->get_contents()?
81:             rm_r(ritor->second);
82:         }
83:         //if not directory, or empty directory, erase
84:         roo_dirents.erase(ritor->first);
85:     }
86:     roo_dirents.erase("."); //erasing root last
87:     roo_dirents.erase(".."); //erasing root last
88:
89: }
90: ////////////// inode_state destructor//////////
91:
92: inode_state::~inode_state(){
93:
94:     rm_r(root);
95:     cwd = nullptr; //need to do this?
96:     root = nullptr; //idk
97: }
98:
99:
100: ostream& operator<< (ostream& out, const inode_state& state) {
101:     //just prints out inode state
102:     //just used in debug statements in working code
103:     out << "inode_state: root = " << state.root//machine addresses
104:         << ", cwd = " << state.cwd;
105:     return out;
106: }
107: //////////////////////////////////////////inode////////////////////////////////////
108: inode::inode(file_type type): inode_nr (next_inode_nr++) {
109:     //constructor
110:     //need a virtual constructor but no such thing in c++
111:     //so instead pass in an argument
```

```
112: //default constructor on an inode has been suppressed
113: //so just say new node and give it the particular filetype that you want to
114: //create
115: //depends on the command it is being called from
116: fileType = type;
117: switch (type) {
118:     case file_type::PLAIN_TYPE:
119:         contents = make_shared<plain_file>();
120:         break;
121:     case file_type::DIRECTORY_TYPE:
122:         contents = make_shared<directory>(); //make shared of a plain
or directory
123:         //adjust the file system
124:         //making filesystem friends
125:         //inode and inode state are already friends
126:         //base file probably doesnt work well without inode
127:         //information hiding is not important?? idk
128:         break;
129:     default: assert (false); //for the sake of clarity
130: }
131: //F ('i', "inode " << inode_nr << ", type = " << type);
132: }
133:
134: size_t inode::get_inode_nr() const {
135:
136:     //just gets inode number, already done
137:     // DEBUGF ('i', "inode = " << inode_nr);
138:     return inode_nr;
139: }
140: //void inode::set_contents(base_file_ptr new_contents){
141: //    contents = new_contents;
142: //} //dont ever need to set new contents though right?
143: base_file_ptr inode::get_contents(){ return contents; } //getter
144:
145: file_type inode::get_file_type(){
146:     cerr<< "filetype: "<< fileType;
147:     return fileType; } //getter need this??
148: //or just use is_dir
149:
150: inode_ptr inode::get_parent(){
151:     return parent;
152: }
153:
154:
155:
156:
157: file_error::file_error (const string& what):
158: //implementation of a file error could have been done in line? idk
159: //need to change?
160:     runtime_error (what) {
161: }
162:
163: //all these functions do is throw a file error based on the file type
164: //that is in basefile
165: //those fucntions will either be inherited or overwritten
166: //if dont override, will be inherited
167:
```



```
168: //can leave alone until plainfile size
169: const wordvec& base_file::readfile() const {
170:
171:     throw file_error ("readfile: is a " + error_file_type());
172: }
173:
174: void base_file::writefile (const wordvec&) {
175:     throw file_error ("writefile: is a " + error_file_type());
176: }
177:
178: void base_file::remove (const string&) {
179:     throw file_error ("remove: is a " + error_file_type());
180: }
181:
182: inode_ptr base_file::mkdir (const string&) {
183:     throw file_error ("mkdir:is a " + error_file_type());
184: }
185:
186: inode_ptr base_file::mkfile (const string&) {
187:     throw file_error ("mkfile:is a " + error_file_type()); //dont work
188: }
189: //added functions
190: map<string,inode_ptr>& base_file::get_dirents() {
191:     throw file_error ("getdirents: is a " + error_file_type()); //dont wo
rk
192: }
193: /*bool base_file::is_dir() {
194:     throw file_error ("isdir: is a " + error_file_type()); //dont work
195: }*/
196:
```

```
// Plainfile must override read and writefile
198: //but can go ahead and inherit remove mkdir and mkfile
199: //all of these need to be done!!!
200: size_t plain_file::size() const { //constant function
201: //use wordvec data
202: //size_t size = data.size(); //does this work
203: // DEBUGF ('i', "size = " << size);
204: return data.size(); //calling size function from map?
205: }
206:
207: const wordvec& plain_file::readfile() const {
208: // DEBUGF ('i', data);
209: return data; //dont change?
210: }
211:
212: void plain_file::writefile (const wordvec& words) {
213: // DEBUGF ('i', words); //must change
214: data = words; //sets data to the wordvec words
215: }
216: /*
217: bool plain_file::is_dir() {
218: return false;
219: }*/
220: //directory must override remove mkdir and mkfile but can inherit
221: //readfile and writefile
222: //all of these need to be done!!!
223:
224: //could just handle plain files initially
225: //because need to make an inode for the root directory
226: //needs to have a directory file in it
227: //but the last three wont be used if dont test using those, dont make or
delete files
228: size_t directory::size() const {
229: //size_t size = dirents.size(); //can use directory.size function in
map?
230: // DEBUGF ('i', "size = " << size);
231: return dirents.size();
232: }
233: //just override the base files
234:
235: void directory::remove (const string& filename) {
236: //DEBUGF ('i', filename); //needs to delete something from a director
Y
237: //idk look at this more
238: //if empty directory or if file
239: //use find() function
240: //shouldnt work on root though? idk
241: inode_ptr rm_ptr = dirents.find(filename)->second;
242: if(rm_ptr->get_file_type() == file_type::PLAIN_TYPE
243: ||dirents.find(filename)->first != ".."){
244: dirents.erase(filename);
245: }
246: }
247:
248: inode_ptr directory::mkdir (const string& dirname) {
249: // DEBUGF ('i', dirname); //creates directory
250: //error if file or directory of same name is already
251: //created, or if the complete pathname to the parent of
```

```
252:    //this dir does not already exist
253:    //dot and dot dot added to dirents
254:
255:    if(dirents.find(dirname)->second == inode_ptr()){ //if it has been c
reated
256:        throw file_error ("mkdir: file already exists: " + dirname); //thr
ow error
257:    }
258:
259:    inode_ptr newDir = make_shared<inode>(file_type::DIRECTORY_TYPE);
260:    //make new dir
261:
262:    //insert new dir to dirents
263:    pair<string,inode_ptr> newPair = {dirname,newDir};
264:    dirents.insert(newPair);
265:
266:    //add dot/dotdot to current dir
267:    pair <string, inode_ptr> dot = {".", newDir}; //sets dot, cwd
268:    (newDir->get_contents()->get_dirents()).insert(dot);
269:
270:    pair <string, inode_ptr> dot_dot = {"..", inode_state().get_cwd()};
//sets dot dot, the paren(cwd before new dir)
271:    (newDir->get_contents()->get_dirents()).insert(dot_dot);
272:
273:    return newDir;
274: }
275:
276: inode_ptr directory::mkfile (const string& filename) {
277:     //DEBUGF ('i', filename); //creates file
278:     //file specified is created and the rest of the wordst
279:     //are put in that file
280:     //if the file already exists, a new one is not created but the
281:     //contents are replaced
282:     //error to specify a directory
283:     //if there are no words the file is empty
284:     inode_ptr i_node_ptr = dirents.find(filename)->second;
285:     if(i_node_ptr->get_file_type() == file_type::DIRECTORY_TYPE){
286:         throw file_error ("mkfile: file is a directory " + filename); /
/throw error
287:     }
288:     //make new file
289:     inode_ptr newFile = make_shared<inode>(file_type::PLAIN_TYPE);
290:     //insert/replace contents
291:     pair<string,inode_ptr> newFilePair = {filename,newFile};
292:     dirents.insert(newFilePair);
293:     return newFile;
294: }
295:
296: map<string,inode_ptr>& directory::get_dirents() {
297:     return dirents;
298: }
299: /*bool directory::is_dir() {
300:     return true;
301: }*/
302: /*inode_ptr directory::get_cwd(){
303:     dirents.find(".")->second;
304: }
305: */
```

```
306: //void directory::writefile (const wordvec&) {
307: //    throw file_error ("writefile: is a " + error_file_type());
308: //}
309: bool directory::file_dne( const string& str){
310:     if(dirents.find(str)== dirents.end()){
311:         return true;
312:     }
313:     return false;
314: }
315: file_type directory::get_file_helper(const string& words){
316:     return dirents.find(words)->second->get_file_type();
317: }
318:
319:
320:
321:
322:
323:
324:
```

```
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 2021-04-30 21:39:38-07 - - $
2:
3: MKFILE      = Makefile
4: DEPFILE     = ${MKFILE}.dep
5: NOINCL      = check lint ci clean spotless
6: NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
7: GMAKE       = ${MAKE} --no-print-directory
8: GPPWARN     = -Wall -Wextra -Wpedantic -Wshadow -Wold-style-cast
9: GPPOPTS     = ${GPPWARN} -fdiagnostics-color=never
10: COMPILECPP  = g++ -std=gnu++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} ${DEPFILE}
44:
45: clean :
46:     - rm ${OBJECTS} ${DEPFILE} core ${EXECBIN}.errs
47:
48: spotless : clean
49:     - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
50:
```

```
51:
52: dep : ${CPPSOURCE} ${CPPHEADER}
53:     @ echo "# ${DEPFILE} created `LC_TIME=C date`" >${DEPFILE}
54:     ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPFILE}
55:
56: ${DEPFILE} : ${MKFILE}
57:     @ touch ${DEPFILE}
58:     ${GMAKE} dep
59:
60: again :
61:     ${GMAKE} spotless dep ci all lis
62: 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 21:39:36 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
```