```
// $Id: commands.h,v 1.11 2016-01-14 14:45:21-08 - - $

#ifndef __COMMANDS_H__
#define __COMMANDS_H__

#include <unordered_map>
using namespace std;

#include "file_sys.h"
#include "util.h"

// A couple of convenient usings to avoid verbosity.

using command_fn = void (*)(inode_state& state, const wordvec& words);
using command_hash = unordered_map<string,command_fn>;

// command_error -
//    Extend runtime_error for throwing exceptions related to this
//    program.

class command_error: public runtime_error {
   public:
      explicit command_error (const string& what);
};

// execution functions -

void fn_cat    (inode_state& state, const wordvec& words);
void fn_cd     (inode_state& state, const wordvec& words);
void fn_echo   (inode_state& state, const wordvec& words);
void fn_exit   (inode_state& state, const wordvec& words);
void fn_ls     (inode_state& state, const wordvec& words);
void fn_lsr    (inode_state& state, const wordvec& words);
void fn_make   (inode_state& state, const wordvec& words);
void fn_mkdir  (inode_state& state, const wordvec& words);
void fn_prompt (inode_state& state, const wordvec& words);
void fn_pwd    (inode_state& state, const wordvec& words);
void fn_rm     (inode_state& state, const wordvec& words);
void fn_rmr    (inode_state& state, const wordvec& words);

command_fn find_command_fn (const string& command);

// exit_status_message -
//    Prints an exit message and returns the exit status, as recorded
//    by any of the functions.

int exit_status_message();
class ysh_exit: public exception {};

#endif

```

```
// $Id: commands.cpp,v 1.26 2021-05-02 19:40:12-07 - - $

#include "commands.h"
#include "debug.h"

command_hash cmd_hash {
   {"cat"   , fn_cat   },
   {"cd"    , fn_cd    },
   {"echo"  , fn_echo  },
   {"exit"  , fn_exit  },
   {"ls"    , fn_ls    },
   {"lsr"   , fn_lsr   },
   {"make"  , fn_make  },
   {"mkdir" , fn_mkdir },
   {"prompt", fn_prompt},
   {"pwd"   , fn_pwd   },
   {"rm"    , fn_rm    },
   {"rmr"   , fn_rmr   },
};

command_fn find_command_fn (const string& cmd) {
   // Note: value_type is pair<const key_type, mapped_type>
   // So: iterator->first is key_type (string)
   // So: iterator->second is mapped_type (command_fn)
  // DEBUGF ('c', "[" << cmd << "]");
   const auto result = cmd_hash.find (cmd);
   if (result == cmd_hash.end()) {
      throw command_error (cmd + ": no such function");
   }
   return result->second;
}

command_error::command_error (const string& what):
            runtime_error (what) {
}

int exit_status_message() {
   int status = exec::status();
   cout << exec::execname() << ": exit(" << status << ")" << endl;
   return status;
}

void fn_cat (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
   //The contents of each file is copied to the standard output. An error is
   //reported if no files are specified, a file does not exist, or is a directory.
   if(words.size() == 1){   //if no files are specified
      throw command_error ("cat: No files are specified"); //dont work
      // program needs to continue!!!!!!
   }
   else{
       wordvec split_path;
        shared_ptr <directory> state_dir = dynamic_pointer_cast<directory>
         (state.get_cwd()->get_contents());
```

```
     // int jcount = 1;
      for(unsigned long j = 1;j < words.size(); ++j){
         //if there is a path, but checks on each call
         split_path = split(words.at(j),"/");//skips make call but includes all paths
        // cout<< "split_path:";
        // cout<< split_path;
        // cout<< "end split path";
         int count =0;
         if(split_path.size()>1){//if its a path
            for(auto i =split_path.begin(); i< split_path.end()-1;i++){//skip last cause creating last so wont exist
               if(state_dir->file_dne(split_path.at(count)) == true){
                  throw command_error ("cat: "+split_path.at(count)+": Directory does not exist");
                  //like foo/wrongdir/bar/newfile
                  //would output wrongdir: no such file or dir
                }
               else if(state_dir->is_dir_(split_path.at(count)) == false){

                  throw command_error ("cat: "+split_path.at(count)+": Directory does not exist");
               }
                count++; //cause the auto is an itr
            }
            //file needs to exist
            if(state_dir->file_dne(words.at(count))==true){
               throw command_error("cat: "+ words.at(j) +": No such file or directory");
            }
            //and not be a directory
            else if(state_dir->is_dir_(words.at(count))==true){
               throw command_error("cat: "+ words.at(j) +": Is a directory");
            }

            cout<< state_dir->get_second(words.at(count))->get_contents()->readfile();//this works
            cout<< "\n";
         }//endif
         //if no path
         else{
            if(state_dir->file_dne(words.at(j))==true){
               throw command_error("cat: "+ words.at(j) +": No such file or directory");
            }
            else if(state_dir->is_dir_(words.at(j))==true){
               throw command_error("cat: "+ words.at(j) +": Is a directory");
            }
            cout<< state_dir->get_second(words.at(j))->get_contents()->readfile();//this works
            cout<< "\n";
         }
      }
   }
//go back to this
}
```

```

void fn_cd (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
}
//dont have to do anything???
void fn_echo (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
   cout << word_range (words.cbegin() + 1, words.cend()) << endl;
}

```

```cpp

void fn_exit (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
   throw ysh_exit();
}

void fn_ls (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
}

void fn_lsr (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
}

void fn_make (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
   shared_ptr <directory> state_dir = dynamic_pointer_cast<directory>
         (state.get_cwd()->get_contents());
   if(words.size()==1){
      throw command_error ("make: No target specified"); //dont work
   }
   wordvec split_path = split(words.at(1),"/");//skips make call but includes all paths
   int count = 0;
   //is a path
   if(split_path.size()>1){//if its a path
      for(auto i =split_path.begin(); i< split_path.end()-1;i++){ //skip last cause creating last so wont exist
         //if file does not exist at all
         if(state_dir->file_dne(split_path.at(count)) == true){
            throw command_error ("make: "+split_path.at(count)+": Directory does not exist");
            //like foo/wrongdir/bar/newfile
            //would output wrongdir: no such file or dir
         }
         //if file exists but is not a directory, needs to be while in the path
         else if(state_dir->is_dir_(split_path.at(count)) == false){
            throw command_error ("make: "+split_path.at(count)+": Directory does not exist");
         }
         count++; //cause the auto is an itr
      }
      //after path, file being made
      if(state_dir->file_dne(split_path.at(count))==false){// if file exists
         if(state_dir->is_dir_(split_path.at(count)) == true){
            throw command_error ("make: "+ words.at(1) +": Is a directory");
         }
         else
         {  //updates file if exists
            inode_ptr updated_pointer = state_dir->update_file(split_path.at(count),wordvec(words.begin()+2,words.end()));
         }
      }
      else{//if file doesnt exist already, makes it
         inode_ptr new_file = state.get_cwd()->get_contents()->mkfile(split_path.at(count));
         //+2 makes it not include make or filename, jsut contents
         new_file->get_contents()->writefile(wordvec(words.begin()+2,words.end()));
      }
   }
   //no path, this works!
   else{
     if(state_dir->file_dne(words.at(1))==true){//this works
         //if file dne then can do normal cause isnt dir or file
         inode_ptr new_file = state.get_cwd()->get_contents()->mkfile(words.at(1));
         //+2 makes it not include make or filename, jsut contents
         new_file->get_contents()->writefile(wordvec(words.begin()+2,words.end()));
      }
      else{//not dir, but updating existing file
         if(state_dir->is_dir_(words.at(1))==true){//this works
             throw command_error ("make: "+ words.at(1) +": Is a directory");
         }
         else{
            inode_ptr updated_pointer = state_dir->update_file(words.at(1),wordvec(words.begin()+2,words.end()));
         // cout<< updated_pointer->get_contents()->readfile();  //makes the file, but includes "make"??
         }
      }
   }
}

void fn_mkdir (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
    if(words.size()==1){
      throw command_error ("mkdir: No target specified"); //dont work
   }
    else if(words.size()>2){
      throw command_error ("mkdir: invalid number of arguments"); //dont work
   }
   wordvec split_path = split(words.at(1),"/");//skips mkdir call but includes all paths
   shared_ptr <directory> state_dir = dynamic_pointer_cast<directory>
         (state.get_cwd()->get_contents());
   //if has path, check if valid
      int count = 0;
      if(split_path.size()>1){//if its a path
         for(auto i =split_path.begin(); i< split_path.end()-1;i++){ //skip last cause creating last so wont exist
            if(state_dir->file_dne(split_path.at(count))==true){//this works //everything needs to exist but the last one
               throw command_error ("mkdir: "+split_path.at(count)+": Directory does not exist"); //dont work
            }
            count++; //cause the auto is an itr
         }
         if(state_dir->file_dne(split_path.at(count)) == false){
            if(state_dir->is_dir_(split_path.at(count)) == true){
               throw command_error ("mkdir: "+split_path.at(count)+": Directory already exists"); //dont work
            }
         }
         else{
            state_dir->mkdir(split_path.at(count));
         }
      }
      //if no path, or end of path, make sure dne
      //should be end if has path or not right?
      else{//if its not a path
         if(state_dir->file_dne(split_path.at(0)) == false){
             if(state_dir->is_dir_(split_path.at(0)) == true){
               throw command_error ("mkdir: "+split_path.at(0)+": Directory already exists"); //dont work
             }
         }
         else{
            state_dir->mkdir(split_path.at(0));
         }
      }
}

void fn_prompt (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
}

void fn_pwd (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
}

void fn_rm (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
}

void fn_rmr (inode_state& state, const wordvec& words) {
   DEBUGF ('c', state);
   DEBUGF ('c', words);
}

```

```cpp
// $Id: debug.h,v 1.12 2019-10-16 15:17:26-07 - - $

#ifndef __DEBUG_H__
#define __DEBUG_H__

#include <bitset>
#include <climits>
#include <string>
using namespace std;

// debug -
//    static class for maintaining global debug flags.
// setflags -
//    Takes a string argument, and sets a flag for each char in the
//    string.  As a special case, '@', sets all flags.
// getflag -
//    Used by the DEBUGF macro to check to see if a flag has been set.
//    Not to be called by user code.

class debugflags {
   private:
      using flagset_ = bitset<UCHAR_MAX + 1>;
      static flagset_ flags_;
   public:
      static void setflags (const string& optflags);
      static bool getflag (char flag);
      static void where (char flag, const char* file, int line,
                         const char* pretty_function);
};

```

```

// DEBUGF -
//    Macro which expands into trace code.  First argument is a
//    trace flag char, second argument is output code that can
//    be sandwiched between <<.  Beware of operator precedence.
//    Example:
//       DEBUGF ('u', "foo = " << foo);
//    will print two words and a newline if flag 'u' is  on.
//    Traces are preceded by filename, line number, and function.

#ifdef NDEBUG
#define DEBUGF(FLAG,CODE) ;
#define DEBUGS(FLAG,STMT) ;
#else
#define DEBUGF(FLAG,CODE) { \
           if (debugflags::getflag (FLAG)) { \
              debugflags::where (FLAG, __FILE__, __LINE__, \
                                 __PRETTY_FUNCTION__); \
              cerr << CODE << endl; \
           } \
        }
#define DEBUGS(FLAG,STMT) { \
           if (debugflags::getflag (FLAG)) { \
              debugflags::where (FLAG, __FILE__, __LINE__, \
                                 __PRETTY_FUNCTION__); \
              STMT; \
           } \
        }
#endif

#endif

```

```
// $Id: debug.cpp,v 1.15 2020-01-22 14:21:55-08 - - $

#include <climits>
#include <iostream>
#include <vector>

using namespace std;

#include "debug.h"
#include "util.h"

debugflags::flagset_ debugflags::flags_ {};

void debugflags::setflags (const string& initflags) {
   for (const unsigned char flag: initflags) {
      if (flag == '@') flags_.set();
                  else flags_.set (flag, true);
   }
}

// getflag -
//    Check to see if a certain flag is on.

bool debugflags::getflag (char flag) {
   // WARNING: Don't TRACE this function or the stack will blow up.
   return flags_.test (static_cast<unsigned char> (flag));
}

void debugflags::where (char flag, const char* file, int line,
                        const char* pretty_function) {
   cout << "DEBUG(" << flag << ") "
        << file << "[" << line << "] " << endl
        << "... " << pretty_function << endl;
}

```

```
// $Id: file_sys.h,v 1.13 2021-05-02 02:03:57-07 - - $

#ifndef __INODE_H__
#define __INODE_H__

#include <exception>
#include <iostream>
#include <memory> //contains shared pointer
#include <map>
#include <vector>
using namespace std;

#include "util.h"

// inode_t -
//    An inode is either a directory or a plain file.

enum class file_type {PLAIN_TYPE, DIRECTORY_TYPE};
class inode;
class base_file;
class plain_file;
class directory;
using inode_ptr = shared_ptr<inode>;
using base_file_ptr = shared_ptr<base_file>;
//implement directory file pointer? mabye not
ostream& operator<< (ostream&, file_type);

```

```

// inode_state -
//    A small convenient class to maintain the state of the simulated
//    process:  the root (/), the current directory (.), and the
//    prompt.

class inode_state {  //only one can exist in the entire filesystem
//inode state does not inherit from inode
//however they are cooperating classes

   friend class inode;
   friend ostream& operator<< (ostream& out, const inode_state&);
   //must be a friend not a member
   private:
      inode_ptr root {nullptr};
      inode_ptr cwd {nullptr};   //need to make a different assignment
      //to them in the constructor!!!
      string prompt_ {"% "};  //cant have the fuction and field name be the same
   public:
      virtual ~inode_state();   //default or make own?
      inode_state (const inode_state&) = delete; // copy ctor
      inode_state& operator= (const inode_state&) = delete; // op=
      //-delete says that if u attempt to copy it the compiler
      //will refuse
      inode_state(); //constructor, have to do some work
      //need to make a destructor?
      //need to make a new inode and point root and cwd at it

      const string& prompt() const;
      //returns prompt when need to print stuff out
      void prompt (const string&);  //implement later?
      //sets the prompts

      inode_ptr get_root();
      inode_ptr get_cwd();
      void set_cwd(inode_ptr);//??? //yes


};

// class inode -
// inode ctor -
//    Create a new inode of the given type.
// get_inode_nr -
//    Retrieves the serial number of the inode.  Inode numbers are
//    allocated in sequence by small integer.
// size -
//    Returns the size of an inode.  For a directory, this is the
//    number of dirents.  For a text file, the number of characters
//    when printed (the sum of the lengths of each word, plus the
//    number of words.(spaces in between words)
//

class inode {

   //inode does not inherit from inode state
   friend class inode_state;
   private:
      static size_t next_inode_nr;  //next inode number
      size_t inode_nr;//inode number itself
      base_file_ptr contents; //basefile is the abstract base class
                           //directory and plainfile as subclasses
                           //dir and pl do all the work
      //inode contains a pointer to base file
      bool is_dir;  //to know the filetype
      inode_ptr parent{nullptr}; //initializes it to null in case there
isnt a parent?
   public:
   virtual ~inode() = default;   //destructor?
      inode (file_type);   //gets filetype or creates filetype?
      size_t get_inode_nr() const;//only copying pointers to inodes
      //hvae to set inode number?
      void set_contents(base_file_ptr);   //setter
      base_file_ptr get_contents(); //getter

      bool isdir(); //getter need this??

      inode_ptr get_parent(); //need these?
      void set_parent(inode_ptr);

};

```

```

// class base_file -
// Just a base class at which an inode can point.  No data or
// functions.  Makes the synthesized members useable only from
// the derived classes.

class file_error: public runtime_error {
   public:
      explicit file_error (const string& what); //its constructor takes in a string
};

class base_file {
   protected:
      base_file() = default;  //basefile has no fieldsl just an abstract class
         //becasue we have constructors defined, we need to specify default
      virtual const string& error_file_type() const = 0;
      //says that this function does not exist in basefile, is an abstract function
      //must be overwritten in a subclass
   public:
      virtual ~base_file() = default;
         //if you dont specify a destructor, it will be specified for you^
         //must declare destructor as virtual, so it doesnt just delete
         //pointers?
      base_file (const base_file&) = delete;
      //movers, not allowing files to be moved
      //means that the implicity generated copier will be prohibited
      //dont want to allow base files to be copied, use POINTERS
      base_file& operator= (const base_file&) = delete;
      virtual size_t size() const = 0;
      //base file has no meaning so it is 0
      //no meaning because can be one thing on a plain and another on a directory file
      virtual const wordvec& readfile() const;
      //will read text from file
      virtual void writefile (const wordvec& newdata);
      //writes text to file, only available to text files
      //error will be thrown for directory file
      virtual void remove (const string& filename);
      //only appropriate if we are dealing with directories?
      virtual inode_ptr mkdir (const string& dirname);
      virtual inode_ptr mkfile (const string& filename);
      virtual map<string,inode_ptr>& get_dirents();
     // virtual bool is_dir();
};
```

```

// class plain_file -
// Used to hold data.
// synthesized default ctor -
//    Default vector<string> is a an empty vector.
// readfile -
//    Returns a copy of the contents of the wordvec in the file.
// writefile -
//    Replaces the contents of a file with new contents.

class plain_file: public base_file {
   //friend ostream?
   private:
      wordvec data;
      virtual const string& error_file_type() const override {
         static const string result = "plain file";
         return result;    //this is ok? no change?
         //if you try and run a function on the wrong type of file,
         //returns the filetype
         //ex remove f , error f is a plainfile
         //static to avoid creating it every time we call a function
      }
   public:
   virtual ~plain_file() = default;
      virtual size_t size() const override;  //must be overridden
      //set size??
      virtual const wordvec& readfile() const override;  //override error functions in base class
      //could make base class functions abstract then require them to be overwritten
      //in base class?
      //should use keyword override
      //cant override something that doesnt exist in the base class
      // would still compile but makes a compile time error so def use override
      //to make it easier to debug
      //keyword virtal is optional
      // virtual void remove (const string& filename)override;
       //virtual inode_ptr mkdir (const string& dirname) override;  //error will be thrown if basefile version is called
                                                                //, implement in directory
      virtual void writefile (const wordvec& newdata) override;
      //virtual inode_ptr mkfile (const string& filename) override;

     // virtual bool is_dir() override;

};

// class directory -
// Used to map filenames onto inode pointers.
// default ctor -
//    Creates a new map with keys "." and "..".
// remove -
//    Removes the file or subdirectory from the current inode.
//    Throws an file_error if this is not a directory, the file
//    does not exist, or the subdirectory is not empty.
//    Here empty means the only entries are dot (.) and dotdot (..).
// mkdir -
//    Creates a new directory under the current directory and
//    immediately adds the directories dot (.) and dotdot (..) to it.
//    Note that the parent (..) of / is / itself.  It is an error
//    if the entry already exists.
// mkfile -
//    Create a new empty text file with the given name.  Error if
//    a dirent with that name exists.

class directory: public base_file {
   private:
      // Must be a map, not unordered_map, so printing is lexicographic
      map<string,inode_ptr> dirents;
      virtual const string& error_file_type() const override {
         static const string result = "directory";
         return result;
      }
   public:
      virtual ~directory() = default;
      virtual size_t size() const override;//????? need to override
      virtual void remove (const string& filename) override;   //need to implement to throw error?
                                                              //if directory name is called
      virtual inode_ptr mkdir (const string& dirname) override;
      virtual inode_ptr mkfile (const string& filename) override;
      //virtual void writefile (const wordvec& newdata) override;//????
       //make a num files
      virtual map<string,inode_ptr>& get_dirents()override;
     // virtual bool is_dir() override;
      inode_ptr get_second(const string& filename);  //need to get rid of?
      bool file_dne(const string& words);
      bool is_dir_(const string& words); //getter need this??
      inode_ptr update_file(const string& filename, const wordvec&words);
};

#endif

```

```
// $Id: file_sys.cpp,v 1.14 2021-05-02 02:03:57-07 - - $

#include <cassert>
#include <iostream>
#include <stdexcept>

using namespace std;

#include "debug.h"
#include "file_sys.h"

size_t inode::next_inode_nr {1}; //initialized to 1, the first inode number

ostream& operator<< (ostream& out, file_type type) {
   switch (type) {
      case file_type::PLAIN_TYPE: out << "PLAIN_TYPE"; break;
      case file_type::DIRECTORY_TYPE: out << "DIRECTORY_TYPE"; break;
      default: assert (false);
   };
   return out;
}




inode_state::inode_state() {
  // DEBUGF ('i', "root = " << root << ", cwd = " << cwd
  //        << ", prompt = \"" << prompt() << "\"");

   //inode state constructor
   //establish inode state
   //create root directory /
   //make sure root directory (parent ..) points at itseld
   //can call inode constructor and pass in a filetype /
   //then modify it after the fact?
   //can call the make shared plain file and directory/?
   //i node and inode state are friends
   //which means once you have an inode the inode state can go in and zap
   //the fields in appropriate manners
   root = make_shared <inode> (file_type::DIRECTORY_TYPE);  //right?
  // shared_ptr <directory> root_dir = dynamic_pointer_cast<directory>
   //                              (root->get_contents());
   //shared_ptr <directory>
   cwd = root;

   pair <string, inode_ptr> dot = {".", root};  //sets dot, cwd, to root
    (root->get_contents()->get_dirents()).insert(dot);

   pair <string, inode_ptr> dot_dot  = {"..", root};  //sets dot dot, the parent to root
    (root->get_contents()->get_dirents()).insert(dot_dot);

}
//inode_state method implementations
void inode_state::prompt(const string& s){
      prompt_  = s;
```

```
}  //implement later? its ok
      //sets the prompts
const string& inode_state::prompt() const { return prompt_; }
//just returns the prompt


inode_ptr inode_state::get_root(){ return root; }

inode_ptr inode_state::get_cwd(){ return cwd; }//need this?

void inode_state::set_cwd(inode_ptr new_cwd){
   cwd = new_cwd;
}



void rm_r( inode_ptr roo){
   //depth first search (postorder)
   map<string,inode_ptr>& roo_dirents = (roo->get_contents()->get_dirent
s());
   //create map of dirents of the file roo
   for(auto ritor = roo_dirents.crbegin(); ritor != roo_dirents.crend();
 ++ritor){ //cr or nah
      //recur over each entry other than dot or dot dot
      if(ritor->first!="." and ritor->first != ".."
         and ritor->second->isdir()
         ==true){//->get_contents()?
         rm_r(ritor->second);
      }
      //if not directory, or empty directory, erase
      roo_dirents.erase(ritor->first);
   }
   roo_dirents.erase("."); //erasing root last
   roo_dirents.erase(".."); //erasing root last

}
/////////// inode_state destructor///////

inode_state::~inode_state(){

   rm_r(root);
   cwd = nullptr;//need to do this?
   root = nullptr;//idk
}


ostream& operator<< (ostream& out, const inode_state& state) {
   //just prints out inode state
   //just used in debug statements in working code
   out << "inode_state: root = " << state.root//machine adresses
       << ", cwd = " << state.cwd;
   return out;
}
/////////////////////////////////inode/////////////////
inode::inode(file_type type): inode_nr (next_inode_nr++) {
   //constructor
   //need a virtual constructor but no such thing in c++
   //so instead pass in anargument
   //default constructor on an inode has been supressed
   //so just say new node and give it the particular filetype that you w
ant to
   //create
   //depends on the command it is being called from
  // fileType = type;
   switch (type) {
      case file_type::PLAIN_TYPE:
         is_dir = false;
           contents = make_shared<plain_file>();
           break;
      case file_type::DIRECTORY_TYPE:
         is_dir = true;
           contents = make_shared<directory>(); //make shared of a plain
 or directory
                //adjust the file sysem
                //making filesystem friends
                //inode and inode state are already friends
                //base file probably doesnt work well without inode
                //information hiding is not important?? idk
           break;
      default: assert (false);   //for the sake of clarity
   }
   //F ('i', "inode " << inode_nr << ", type = " << type);
}

size_t inode::get_inode_nr() const {

   //just gets inode number, already done
  // DEBUGF ('i', "inode = " << inode_nr);
   return inode_nr;
}
//void inode::set_contents(base_file_ptr new_contents){
//   contents = new_contents;
//}   //dont ever need to set new contents though right?
base_file_ptr inode::get_contents(){ return contents; } //getter

bool inode::isdir(){
   return is_dir; } //getter need this??
//or just use is_dir

inode_ptr inode::get_parent(){
   return parent;
}




file_error::file_error (const string& what):
//implementation of a file error could have been done in line? idk
//need to change?
            runtime_error (what) {
}

//all these functions do is throw a file error based on the file type
//that is in basefile
   //those fucntions will either be inherited or overwritten
   //if dont override, will be inherited
```

```

//can leave alone until plainfile size
const wordvec& base_file::readfile() const {

   throw file_error ("readfile: is a " + error_file_type());
}

void base_file::writefile (const wordvec&) {
   throw file_error ("writefile: is a " + error_file_type());
}

void base_file::remove (const string&) {
   throw file_error ("remove: is a " + error_file_type());
}

inode_ptr base_file::mkdir (const string&) {
   throw file_error ("mkdir:is a " + error_file_type());
}

inode_ptr base_file::mkfile (const string&) {
   throw file_error ("mkfile:is a " + error_file_type()); //dont work
}
//added functions
map<string,inode_ptr>& base_file::get_dirents() {
   throw file_error ("getdirents: is a " + error_file_type()); //dont wo
rk
}
/*bool base_file::is_dir() {
   throw file_error ("isdir: is a " + error_file_type()); //dont work
}*/

```

```
//plainfile must override read and writefile
   //but can go ahead and inherit remove mkdir and mkfile
//all of these need to be done!!!
size_t plain_file::size() const {   //constant function
   //use wordvec data
   //size_t size  = data.size();  //does this work
  // DEBUGF ('i', "size = " << size);
   return data.size();  //calling size function from map?
}

const wordvec& plain_file::readfile() const {
  // DEBUGF ('i', data);
   return data;   //dont change?
}

void plain_file::writefile (const wordvec& words) {
  // DEBUGF ('i', words);//must change
   data = words;  //sets data to the wordvec words
}
/*
bool plain_file::is_dir() {
   return  false;
}*/
//directory must override remove mkdir and mkfile but can inherit
//readfile and writefile
//all of these need to be done!!!

//could just handle plain files initially
//because need to make an inode for the root directory
//needs to have a directory file in it
//but the last three wont be used if dont test using those, dont make or
 delete files
size_t directory::size() const {
   //size_t size = dirents.size();  //can use directory.size function in
 map?
  // DEBUGF ('i', "size = " << size);
   return dirents.size();
}
//just override the base files

void directory::remove (const string& filename) {
   //DEBUGF ('i', filename); //needs to delete something from a director
y
   //idk look at this more
   //if empty directory or if file
   //use find() function
   //shouldnt work on root though? idk
     inode_ptr rm_ptr = dirents.find(filename)->second;
   if(rm_ptr->isdir() == false
       ||dirents.find(filename)->first != ".."){
      dirents.erase(filename);
   }
}

inode_ptr directory::mkdir (const string& dirname) {
  // DEBUGF ('i', dirname);//creates directory
   //error if file or directory of same name is already
   //created, or if the complete pathname to the parent of
   //this dir does not already exist
   //dot and dot dot added to dirents

   if(dirents.find(dirname)->second == inode_ptr()){  //if it has been c
reated
      throw file_error ("mkdir: file already exists: " + dirname); //thr
ow error
   }

   inode_ptr newDir = make_shared<inode>(file_type::DIRECTORY_TYPE);
   //make new dir

   //insert new dir to dirents
   pair<string,inode_ptr> newPair = {dirname,newDir};
   dirents.insert(newPair);

   //add dot/dotdot to current dir
   pair <string, inode_ptr> dot = {".", newDir};  //sets dot, cwd
   (newDir->get_contents()->get_dirents()).insert(dot);

   pair <string, inode_ptr> dot_dot  = {"..", inode_state().get_cwd()};
 //sets dot dot, the paren(cwd before new dir)
   (newDir->get_contents()->get_dirents()).insert(dot_dot);

   return newDir;
}

inode_ptr directory::mkfile (const string& filename) {
   //DEBUGF ('i', filename); //creates file
//file specified is created and the rest of the wordst
//are put in that file
//if the file already exists, a new one is not created but the
//contents are replaced
//error to specify a directory
//if there are no words the file is empty
   //inode_ptr i_node_ptr = dirents.find(filename)->second;
   /*if(i_node_ptr->isdir() == true){
          throw file_error ("mkfile: file is a directory " + filename); /
/throw error
   }*/
   //make new file
   inode_ptr newFile = make_shared<inode>(file_type::PLAIN_TYPE);
   //insert/replace contents
   pair<string,inode_ptr> newFilePair = {filename,newFile};
   dirents.insert(newFilePair);//dirents[filename]= newFile;
   return newFile;
}

map<string,inode_ptr>& directory::get_dirents() {
   return dirents;
}
/*bool directory::is_dir() {
   return true;
}*/
/*inode_ptr directory::get_cwd(){
   dirents.find(".")->second;
}
*/
```

```
//void directory::writefile (const wordvec&) {
//   throw file_error ("writefile: is a " + error_file_type());
//}
bool directory::file_dne( const string& str){
   if(dirents.find(str)== dirents.end()){
      return true;
   }
   return false;
}
bool directory::is_dir_(const string& words){
  return( dirents.find(words)->second->isdir());
}

inode_ptr directory::update_file(const string& filename, const wordvec& 
words){
   inode_ptr update_ptr = dirents.find(filename)->second;
   update_ptr->get_contents()->writefile(words);
   pair<string,inode_ptr> update_pair = {filename,update_ptr};
   dirents.insert(update_pair);//dirents[filename]= newFile;
   return update_ptr;
}
inode_ptr directory::get_second(const string& filename){
   return dirents.find(filename)->second;
}





```

```
// $Id: util.h,v 1.14 2020-10-22 18:00:02-07 - - $

// util -
//    A utility to provide various services not conveniently
//    included in other modules.

#ifndef __UTIL_H__
#define __UTIL_H__

#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>
using namespace std;

// Convenient type using to allow brevity of code elsewhere.

template <typename iterator>
using range_type = pair<iterator,iterator>;

using wordvec = vector<string>;
using word_range = range_type<decltype(declval<wordvec>().cbegin())>;

// want_echo -
//    We want to echo all of cin to cout if either cin or cout
//    is not a tty.  This helps make batch processing easier by
//    making cout look like a terminal session trace.

bool want_echo();

//
// main -
//    Keep track of execname and exit status.  Must be initialized
//    as the first thing done inside main.  Main should call:
//       main::execname (argv[0]);
//    before anything else.
//

class exec {
   private:
      static string execname_;
      static int status_;
      static void execname (const string& argv0);
      friend int main (int, char**);
   public:
      static void status (int status);
      static const string& execname() {return execname_; }
      static int status() {return status_; }
};

```

```

// split -
//    Split a string into a wordvec (as defined above).  Any sequence
//    of chars in the delimiter string is used as a separator.  To
//    Split a pathname, use "/".  To split a shell command, use " ".

wordvec split (const string& line, const string& delimiter);

// complain -
//    Used for starting error messages.  Sets the exit status to
//    EXIT_FAILURE, writes the program name to cerr, and then
//    returns the cerr ostream.  Example:
//       complain() << filename << ": some problem" << endl;

ostream& complain();

// operator<< (vector) -
//    An overloaded template operator which allows vectors to be
//    printed out as a single operator, each element separated from
//    the next with spaces.  The item_t must have an output operator
//    defined for it.

template <typename item_t>
ostream& operator<< (ostream& out, const vector<item_t>& vec) {
   string space = "";
   for (const auto& item: vec) {
      out << space << item;
      space = " ";
   }
   return out;
}

template <typename iterator>
ostream& operator<< (ostream& out, range_type<iterator> range) {
   for (auto itor = range.first; itor != range.second; ++itor) {
      if (itor != range.first) out << " ";
      out << *itor;
   }
   return out;
}

#endif

```

```
// $Id: util.cpp,v 1.14 2019-10-08 14:01:38-07 - - $

#include <cstdlib>
#include <unistd.h>

using namespace std;

#include "util.h"
#include "debug.h"

bool want_echo() {
   constexpr int CIN_FD {0};
   constexpr int COUT_FD {1};
   bool cin_is_not_a_tty = not isatty (CIN_FD);
   bool cout_is_not_a_tty = not isatty (COUT_FD);
   DEBUGF ('u', "cin_is_not_a_tty = " << cin_is_not_a_tty
          << ", cout_is_not_a_tty = " << cout_is_not_a_tty);
   return cin_is_not_a_tty or cout_is_not_a_tty;
}

string exec::execname_; // Must be initialized from main().
int exec::status_ = EXIT_SUCCESS;

string basename (const string &arg) {
   return arg.substr (arg.find_last_of ('/') + 1);
}

void exec::execname (const string& argv0) {
   execname_ = basename (argv0);
   cout << boolalpha;
   cerr << boolalpha;
   DEBUGF ('u', "execname = " << execname_);
}

void exec::status (int status) {
   if (status_ < status) status_ = status;
}

```

```

wordvec split (const string& line, const string& delimiters) {
   wordvec words;
   size_t end = 0;

   // Loop over the string, splitting out words, and for each word
   // thus found, append it to the output wordvec.
   for (;;) {
      size_t start = line.find_first_not_of (delimiters, end);
      if (start == string::npos) break;
      end = line.find_first_of (delimiters, start);
      words.push_back (line.substr (start, end - start));
   }
   DEBUGF ('u', words);
   return words;
}

ostream& complain() {
   exec::status (EXIT_FAILURE);
   cerr << exec::execname() << ": ";
   return cerr;
}

```

```
// $Id: main.cpp,v 1.12 2021-04-30 22:15:28-07 - - $

#include <cstdlib>
#include <iostream>
#include <string>
#include <utility>
#include <unistd.h>

using namespace std;

#include "commands.h"
#include "debug.h"
#include "file_sys.h"
#include "util.h"

// scan_options
//    Options analysis:  The only option is -Dflags.

void scan_options (int argc, char** argv) {
   opterr = 0;
   for (;;) {
      int option = getopt (argc, argv, "@:");
      if (option == EOF) break;
      switch (option) {
         case '@':
            debugflags::setflags (optarg);
            break;
         default:
            complain() << "-" << static_cast<char> (option)
                       << ": invalid option" << endl;
            break;
      }
   }
   if (optind < argc) {
      complain() << "operands not permitted" << endl;
   }
}

```

```

// main -
//    Main program which loops reading commands until end of file.

int main (int argc, char** argv) {
   exec::execname (argv[0]);
   cout << boolalpha;  // Print false or true instead of 0 or 1.
   cerr << boolalpha;
   cout << argv[0] << " build " << __DATE__ << " " << __TIME__ << endl;
   scan_options (argc, argv);
   bool need_echo = want_echo();
   inode_state state;
   try {
      for (;;) {
         try {
            // Read a line, break at EOF, and echo print the prompt
            // if one is needed.
            cout << state.prompt();
            string line;
            getline (cin, line);
            if (cin.eof()) {
               if (need_echo) cout << "^D";
               cout << endl;
               DEBUGF ('y', "EOF");
               break;
            }
            if (need_echo) cout << line << endl;

            // Split the line into words and lookup the appropriate
            // function.  Complain or call it.
            wordvec words = split (line, " \t");
            DEBUGF ('y', "words = " << words);
            //if(words.size()>1){
               string word0 = words.at(0);
               if(!(word0.at(0)=='#')){
                  command_fn fn = find_command_fn (words.at(0));
                  fn (state, words);
               }
           // }
         }catch (file_error& error) {
            complain() << error.what() << endl;
         }catch (command_error& error) {
            complain() << error.what() << endl;
         }
      }
   } catch (ysh_exit&) {
      // This catch intentionally left blank.
   }

   return exit_status_message();
}

```

```
# $Id: Makefile,v 1.41 2021-04-29 21:33:33-07 - - $

MKFILE      = Makefile
DEPFILE     = ${MKFILE}.dep
NOINCL      = check lint ci clean spotless
NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
GMAKE       = ${MAKE} --no-print-directory
GPPWARN     = -Wall -Wextra -Wpedantic -Wshadow -Wold-style-cast
GPPOPTS     = ${GPPWARN} -fdiagnostics-color=never
COMPILECPP  = g++ -std=gnu++17 -g -O0 ${GPPOPTS}
MAKEDEPCPP  = g++ -std=gnu++17 -MM ${GPPOPTS}

MODULES     = commands debug file_sys util
CPPHEADER   = ${MODULES:=.h}
CPPSOURCE   = ${MODULES:=.cpp} main.cpp
EXECBIN     = yshell
OBJECTS     = ${CPPSOURCE:.cpp=.o}
MODULESRC   = ${foreach MOD, ${MODULES}, ${MOD}.h ${MOD}.cpp}
OTHERSRC    = ${filter-out ${MODULESRC}, ${CPPHEADER} ${CPPSOURCE}}
ALLSOURCES  = ${MODULESRC} ${OTHERSRC} ${MKFILE}
LISTING     = Listing.ps

export PATH := ${PATH}:/afs/cats.ucsc.edu/courses/cse110a-wm/bin

all : ${EXECBIN}

${EXECBIN} : ${OBJECTS}
        ${COMPILECPP} -o $@ ${OBJECTS}

%.o : %.cpp
        - checksource $<
        - cpplint.py.perl $<
        ${COMPILECPP} -c $<

ci : check
        - cid -is ${ALLSOURCES}

check : ${ALLSOURCES}
        - checksource ${ALLSOURCES}
        - cpplint.py.perl ${CPPSOURCE}

lis : ${ALLSOURCES}
        mkpspdf ${LISTING} ${ALLSOURCES} ${DEPFILE}

clean :
        - rm ${OBJECTS} ${DEPFILE} core ${EXECBIN}.errs

spotless : clean
        - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}

```

```

dep : ${CPPSOURCE} ${CPPHEADER}
        @ echo "# ${DEPFILE} created `LC_TIME=C date`" >${DEPFILE}
        ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPFILE}

${DEPFILE} : ${MKFILE}
        @ touch ${DEPFILE}
        ${GMAKE} dep

again :
        ${GMAKE} spotless dep ci all lis
submit:
        submit cse111-wm.s21 asg2 ${ALLSOURCES} README

ifeq (${NEEDINCL}, )
include ${DEPFILE}
endif

```

```
# Makefile.dep created Sun May  2 19:40:11 PDT 2021
commands.o: commands.cpp commands.h file_sys.h util.h debug.h
debug.o: debug.cpp debug.h util.h
file_sys.o: file_sys.cpp debug.h file_sys.h util.h
util.o: util.cpp util.h debug.h
main.o: main.cpp commands.h file_sys.h util.h debug.h
```