PURPOSE: In this program we are creating a simple shell. It uses system calls to carry out commands given on a command line.

REFERENCE:
https://users.soe.ucsc.edu/~sbrandt/111/code/DESIGN

Pipe	ls | wc	working
Infinite Pipe	ls | grep ".c" | wc	working
Redirect out	grep ".c" . > grep.out	working
Redirect in	cat < grep.out	working

Operations:
int main (void)
Description: An infinite while loop runs. At every iteration, it waits for a string to be entered into our shell. It processes the line by tokenizing it with flex. It send every command entered into the function parseargs, which is where everything starts off.
INPUT: None
OUTPUT: int
ALGORITHM: The logic of the program begins within a simple while loop, that continually waits for lines to be read in. A line is read in using the getline() function, which utilizes flex to tokenize the string. If the line is not empty, the line read in is passed into a function which will use analyze the contents of the command called parseargs and start the next phase of the program: executing the command.

char *concat(char* a, char* b);
Description: Takes in two character arrays, concatenates them and returns a new character array.
INPUT: two strings a and b
OUTPUT: one string a+b
ALGORITHM: malloc memory equal to the length of both strings, copy both strings into this new memory using strcpy. Return a pointer to the new memory location.

char *which(char* cmd);

This beautiful function calls $which <command> on the given command and returns a string containing the exact path, this is super useful for all the exec family of functions.
INPUT: a command (e.g “ls”)
OUTPUT: a path to input command (e.g “/bin/ls”)
ALGORITHM:

1) open a pipe
2) fork the process
3) child execs “which” given the input command with stdout redirected to the pipe
4) parent waits on child
5) when child dies the parent reads from the pipe to a buffer and returns a pointer to the buffer.

void shell_pipe2(char** command, int save[]);

Takes in a command and executes it, using stdin as input to the command. The output of this command is placed into stdin for the next command to use as input. This function is only executed if a pipe has been encountered and we must process a command that uses the contents of stdin. Before entering this function, we have already processed another command and put its contents into stdin, and the command executed in this function is utilizing that.
INPUT: a command, file pointers to the location of stdin
OUTPUT: nothing
RESULT: we are able to do infinite pipes
ALGORITHM:

1. The algorithm gets the path to the command passed in by using the “which” function.
2. A pipe is created passing int fd[2] into it. This is going to allow a child and a parent process to talk to each other.
3. The process is forked. A process id is returned. This will let us know whether we are dealing with a child or a parent process.
4. If it is a child process, the file descriptor for reading is closed (fd[0]), since we will only be writing right now. Then, we dup stdout so that the output we are about the use goes into a file descriptor we have access to. This will be important later. Then the command is executed using execv. The command’s path is given as the first argument. The second argument is a char** containing the command and all its flags. Since we have previously duped stdin before entering into this method, the command reads from the contents of the command that must have been executed.
5. If the process id is a parent, then we close the file descriptor for reading. And then we dup standard input. The file descriptor is passed in so that the new stdin is actually the contents of our command. Then we wait for the child process to die. Once it has died, we save the file descriptors to int save[2], so that we can access the file later on.

void redirect_output(char** LHS, char* filename);
The void function redirects the output of a command from stdout to a file.
INPUT: a shell command (e.g “ls -l”) and a filename (e.g “ls.txt”)
RESULT: file with given filename is created. contents of file are the result of running the input command.
ALGORITHM:
The algorithm first opens the filename that is passed to it. We then find out the path the the command using the which() command located in the program. Then, we fork the process to create a child and a parent process. The fork will return a process id. If it is 0, then we know that it is a child process. If it is not, then we know that it is a parent process. The child process will execute the command that we want. Before we do this, we clear stdout by flushing it. Then, we use the command dup2 to replace stdout with the file contents. The first argument of dup2 is the file number assigned to the file we had opened, the second argument is 1 - this signifies that we are closing stdout and replacing it with the file. Then, we use execv to execute the command. We pass in the command’s path as the first argument and in the second argument, we pass the entire char** that contains the contents of the command, including its arguments.

void print_array(char ** array, char* caller);
Description: Prints the contents of an array of strings. This function is for debugging
INPUT: array of strings (e.g [“ab”, “cd”, “ef”])
OUTPUT: prints the contents to stdout
ALGORITHM: iterates over an array printing the element at each index.

int array_length(char** array);
We found that having the length of a string array would be useful, this doesn’t exit in C, so we wrote it.
INPUT: an array: [0,1,2,3,4]
OUTPUT: the length of input array: 5
ALGORITHM: start a counter with a value of 0. iterate over the array incrementing a counter, when you reach a NULL value in the array, return the counter.

int begin_first_cmd(char** args);
Description: Determines whether or not the first command encountered should be executed or not before entering into the switch statement and iterating over the rest of the arguments.
INPUT: The entire command that the user has typed in is passed in. No temporary null pointers are inserted into the command. If the command with its options directly precedes a pipe or a null pointer, then a 1 will be returned, representing “Yes”. Or else, a 0 will be returned, representing “No”.
OUTPUT: integer representing whether or not a command should be executed. 1 for yes, 0 for no.
RESULT: We are able to know when to correctly start off a chain of pipes or simply execute a single command.

void standard_exec(char** command, int save[], int original[]);
Description: Takes in and executes a command the user has typed in that precedes zero or one pipe. Note that therefore, this function will not be executed if it precedes a redirect in or out. Because this is the first command executed given the constraints, it will also save the original state of stdio and stdout as file descriptors. These will be used later when closing up the program.
INPUT: a char** containing a command and its options
OUTPUT: none
RESULT: a command is executed that may possibly set of a chain of zero to infinite pipes. The original state of stdio and stdout are also saved and can be used later to close up the command the user typed in.
ALGORITHM:

1. We find out the path the the command using the which() command located in the program.
2. Then, we create a pipe so that we can replace stdout and stdin with the output of our command.
3. Fork the process to create a child and a parent process. The fork will return a process id. If it is 0, then we know that it is a child process. If it is not, then we know that it is a parent process.
4. The child process will execute the command that we want. Before we do this, we dup stdout with our file descriptor. This is going to force the output of the command to be located at the contents of the file descriptor. Then, we use execv to execute the command. We pass in the command's path as the first argument and in the second argument, we pass the entire char** that contains the contents of the command, including its arguments.
5. The parent process will wait for the child process to die. Then when it dies, it will dup stdin with the file descriptor at index 0. Now we have a hook to stdin too with the file descriptor. Then we wait for the child process to die.

int get_cmd_end(char** RHS_start);
Description: Checks an array the first instance of a special character and returns an index to that character.
INPUT: argument array.
OUTPUT: index of first special character, if no character exists return the index of the null terminating element.
RESULT: This is going to give us information about where a command of interest ends. The command and its arguments are each stored in their own index in char** args. If char** is passed in, this will tell us where the end of this command and its arguments are by passing an integer back representing where it ends in char**.
OUTPUT: The index of where a command in a char** ends
ALGORITHM:

Start a counter with a value of 0. iterate over the array incrementing a counter, when you reach a NULL value, or a pipe ('|'), or a redirect in ('<'), or a redirect out ('>') in the array, return the counter.

void redirect_input(char** LHS, char* filename);
Description: Redirect input to be from a file instead of stdin.
INPUT: command with arguments, and filename.
RESULT: command is run with input redirected to be from the given file.
OUTPUT: none
ALGORITHM:
The algorithm first opens the filename that is passed to it. We then find out the path the the command using the which() command located in the program. Then, we fork the process to create a child and a parent process. The fork will return a process id. If it is 0, then we know that it is a child process. If it is not, then we know that it is a parent process. The child process will execute the command that we want. Before we do this, we clear stdin by flushing it. Then, we use the command dup2 to replace stdin with the file contents. The first argument of dup2 is the file number assigned to the file we had opened, the second argument is 0 - this signifies that we are closing stdin and replacing it with the file. Then, we use execv to execute the command. We pass in the command's path as the first argument and in the second argument, we pass the entire char** that contains the contents of the command, including its arguments.

void parseargs(char** args);
Description: The main meat of the shell, parses arguments and handles special command cases. This function is called upon every successful return from getlines in main.
OUTPUT: none
ALGORITHM:

1) If the first argument must be evaluated immediately (in the case of a single command or pipe) then evaluate it. Store the result in a stream.
2) Begin to examine the array of commands, go command by command looking for special characters when we find a special character we take an action and replace it with a null terminating symbol.
    a) if you find a pipe, parse out the left hand and right hand sides of the pipe and send them off to the shell_pipe subroutine.
    b) if you find a redirect parse out the left hand side and the filename and send them off to the redirect subroutines
    c) if you find a special command (e.g exit) go ahead and process it directly.
3) If the result from any above execution needs to be printed it will be saved in a file descriptor
    a) if we are in this case, open the file descriptor and print it to stdout,
4) Clean up all saved file descriptors and restore stdin and stdout.

int prefix_strcmp(char* arg, char* match);
Description: Takes in two char*s. One is called ‘arg’ and another is called ‘match’. The function checks to see if ‘arg’’s prefix is ‘match’.
INPUT: two char*s
OUTPUT: integer representing whether the prefix of the first char* equals the second char*
RESULT: an integer representing whether or not they match. If they match, it will return a 1. If they do not match, it will return a 0.
ALGORITHM:
Iterate over all the characters in both strings, where the same index is checked in both strings at the same time. At every iteration, we check to see if the characters in the string have the same ASCII value. If they do not, they are not the same character - immediately a 0 will be returned. As soon as one of them reaches the null pointer, exit the iteration loop. Using the same index that caused the loop to exit, check if the string ‘match’ contains a null pointer at this location. If it does, then this means that you have iterated through the entire string without finding a mismatch, therefore the string called ‘args’ has a prefix that is identical to ‘match’. The int 1 will be returned at this point.

void strict_exec(char** args);
Execute a command without forking
INPUT: command with arguments
OUTPUT: none
RESULT: command is executed without a fork
ALGORITHM: perform the which subroutine on args[0], this returns the command. Then exec the command and arguments without forking.