Project 3: A Simple Bash-Like Shell

In this project you and your teammate will implement a simple shell called `clash` ("class shell"). The behavior of `clash` is almost identical to `bash`, but it only supports a subset of the features in `bash`. If you're not sure how a particular feature should work, try the same feature in `bash`. The intent is that the descriptions below all match bash behavior; if you find that they don't, let me know and implement the bash behavior instead. Note: the manual page for `bash` is not totally correct in its descriptions of how `bash` works, so use `bash`, not its manual page, as your reference point.

Overview

`Clash` executes *commands*, where each command consists of one or more *words*. `Clash` reads input and parses the input according to the syntax rules described below to produce the words for a command. Then it executes the command. Typically, the first word of a command selects an executable file; `clash` spawns a new process to execute that file and passes all of the words to the process as its `argv` arguments. Some commands are executed directly by `clash` (*built-in commands*), so they do not result in the creation of a new process. Processes can be grouped into *pipelines*, where data flows from one process to the next in the pipeline.

Syntax

Command separators: commands are separated by semicolons (";") and/or newline characters. The pipeline character ("|") also serves as a command separator.

Word separators: space and tab characters are treated as word separators; newlines are also treated as word separators during variable and command substitution. The characters ";", "<", ">", and "|" also act as word separators.

Backslash substitution: a backslash *quotes* the character that follows it, so that the following character becomes part of the current word and receives no special treatment; the backslash is dropped. For example, backslash followed by a space character causes the space character to appear in the current word; it does not act as a word separator. "\$" causes "$" to be inserted in the current word; it will not trigger variable substitution. If a backslash character is followed by a newline, then both the backslash and the newline will be dropped; neither will appear in the word, and the newline is not treated as a command separator.

Single quotes: when a single-quote character appears, all of the characters between that single-quote and the next single-quote will be incorporated literally into the current word; none of these characters will be given any special treatment (i.e. backslashes inside single-quotes do not invoke backslash substitution; they will appear in the word). The single quotes do not appear in the word.

Double quotes: when a double-quote character appears, all of the characters between that double-quote and the next unquoted double-quote will be incorporated into the current word. Command substitution, variable substitution, and backslash substitution are performed on the information between double quotes, but no other characters are treated specially. For example, spaces are not treated as word separators and ";" is not treated as a command separator. Backslash substitution between double quotes is slightly different then described above: backslash substitution is only performed if the character following the backslash is "$", "`" (backquote), double-quote, backslash, or newline; in all other cases, the backslash and the following character are simply copied to the current word.

Variable substitution: when a dollar sign appears ("$"), it triggers variable substitution. The characters following the dollar sign are treated as the name of a variable, and the value of that variable is inserted in the current word in place of the name. The variable name can take three forms:

- If the character after the $ is a letter, then the variable name consists of all characters up to the first one that is neither a letter nor a digit.
- If the character after the $ is a digit or one of the characters "*", "#", or "?", then the variable name is that one character.
- If the character after the $ is a "{" (open brace) then the variable name consists of all the characters between that open brace and the next "}" (close brace). No substitutions are performed on the characters between the braces.

If there is no variable by the given name, then `clash` should behave as if the variable existed with an empty value. White space in the value of a variable (spaces, tabs, or newlines) results in additional word breaks. This process is described in more detail below.

Command substitution: when a backquote character appears ("`` ` ``"), it triggers command substitution. The characters between the backquote character and the next unquoted backquote character are treated as a shell script (which can contain one or more pipelines) and executed. The standard output generated by this execution is captured and substituted in the current word in place of the command. While scanning for the terminating backquote character, no substitutions are performed except that "``\` ``" is replaced with "`` ` ``". Once the closing backquote has been found and the command is executed, the result is copied into the current word and reparsed from scratch; at this point, all of the usual substitutions will occur. When the result of the command is copied into the current word, spaces, tabs, and newlines result in additional word breaks. This process is described in more detail below.

Tilde substitution: if the first character of a word is a "~", it triggers tilde substitution. All of the characters up to the next "/" or the end of the word, whichever comes first, are treated as a user name; if there exists a user by that name, then path for that user's home directory is substituted in place of the tilde and the name. If no such user exists, then no substitution is performed: the tilde and the user name are passed through to the current word. If the user name is empty, then the value of the HOME variable is substituted.

Path expansion: if a word contains any of the unquoted characters "?", "*", or "[", then path expansion is performed: `clash` will treat the contents of the word as a pattern, find the names of all existing files that match that pattern, and substitute those file names for the original word. Each matching file name becomes a separate word. "?" matches any single character in a file's name. "*" matches any contiguous range of characters. For example, the pattern "*/*.cc" will match all files in all subdirectories of the current directory that have a ".cc" extension. Brackets, such as "[xyz]", will match one character if it appears between the brackets. If the first character between the brackets is a "^", such as "[^xyz]", then the sense is reversed: a character will be matched only if it *doesn't* appear between the brackets. If the characters between brackets include a structure such as "a-c", it is equivalent to listing all of the characters in that range. For example, "[0-9]" matches any digit. If the pattern does not match any files, then the pattern itself is copied to the current word.

Pipelines and Redirection

If two commands are separated by "|" characters, then they form a pipeline: the standard output of the first command is written to a pipe, which serves as the standard input for the second command. A pipeline may contain any number of commands.

If an unquoted ">" character appears in a command, then the word following the ">" is treated as the name of a file and the standard output of the command will be redirected to that file. If the command is part of a pipeline, then the redirection applies to that command; this means that the standard input for the following command in the pipeline will be empty (i.e. immediate end-of-file condition). If an unquoted "<" character appears in a command, then the following word is treated as the name of a file and the standard input of the command will be read from that file. If the command is part of a pipeline, then any output from the previous pipeline stage will be discarded.

Parsing Order

The syntax rules described above are fairly simple when considered in isolation, but they combine in ways that create a lot of complexity. The exact order in which the rules are applied is crucial to the behavior of `clash`. Your implementation of `clash` must behave as if the substitutions are performed in the following order (your implementation needn't necessarily follow this order, but it must produce the same results as if it did):

- Command breaking: divide the input into pipelines and commands. Also, identify I/O redirection and separate out the words containing the file names for redirection. These words are expanded and substituted using the same steps described below for the rest of the command.
- Tilde substitutions. Since variable substitutions haven't been performed yet, the name of the user can't come from a variable, and new words introduced by variable and command substitution will not undergo tilde substitutions.
- A single left-to-right scan to process backslashes, single quotes, double quotes, variable substitutions, and command substitutions.
- Word breaking. Since this is performed after variable and command substitution, white space introduced by these substitutions will result in new words. It is also possible for words to disappear in this step. For example, if variable `x` has an empty value, then the command

  ```
  echo $x $x
  ```

  will have only a single word ("echo"). However, a word cannot be removed if it originally contained single or double quotes. For example, the command

  ```
  echo "" ""
  ```

  will have three words, of which the last two will be empty. When breaking words in this step, white space that has been quoted (it appeared in single quotes or double quotes, or it was backslashed) does not trigger word breaks. For example, the command

  ```
  echo 'a b c'
  ```

  has only two words.
- Path expansion. The pattern matching characters "?", "*", and "[" may come from variable values or command substitutions, but if any of these characters were quoted, then they are not treated as special for pattern matching (this also means that if a variable is substituted inside double quotes, a "*" character in that variable's value is not treated as special). Path expansion can introduce additional words.

Command Execution

Once a command has been parsed into words and all of the substitutions and expansions have been performed, the command is executed. In the normal case this is done by creating one subprocess for each command, using `fork` and `exec`. If the first word of a command contains a "/" character, then the contents of that word are used as the name of the executable file; otherwise, the PATH variable is used to select the executable as described below. Between the calls to `fork` and `exec` for each subprocess, you must modify file descriptors for the child in order to implement I/O redirection and pipelines. The environment variables passed to each child must reflect `clash`'s variable bindings, as discussed below.

There are two situations in which `clash` executes a command directly, rather than spawning a subprocess: variable assignments and built-in commands. If a command contains a single word, and if the word starts with a valid variable name (an initial letter followed by any number of letters or digits) followed by "=", then the text following the "=" is used as the new value for the given variable. In this case, `clash` performs the variable assignment instead of creating a subprocess. All the usual substitutions are performed before executing a variable assignment. For example,

```
output="`wc *.cc`"
```

will set the value of the variable `output` to the standard output produced by the `wc` command.

In addition to variable assignments, `clash` supports a collection of *builtin commands*, which it executes directly. Here are the builtin commands you must support:

`cd` *dirName*
Changes the current working directory of the shell, using *dirName* as the path to the new working directory. If no directory name is given, the value of the HOME variable is used as the new working directory.

`exit` *status*
Causes `clash` to exit, returning *status* as its exit status. If no *status* argument is provided, then the exit status will be 0.

`export` *varName varName varName* ...
Each argument is the name of a variable. If a variable exists by that name, it is marked for export, which means that it will be passed to subprocesses as an environment variable.

`unset` *varName varName varName* ...
Each argument is the name of a variable. If a variable exists by that name, it is deleted.

Reading Commands

If `clash` is invoked with no command-line arguments, it reads commands from its standard input, which can be either a terminal or a file. If standard input is a terminal (which can be determined with the `isatty` function), then `clash` outputs a "% " prompt before reading each line of input. In most cases, each line of input can be parsed and executed separately: it contains either a single command, a pipeline, or a collection of commands or pipelines separated by semicolons. In this case, clash parses and executes each line of input before reading the next line.

However, there may be times when a line of input does not contain a complete command, such as the following:

```
echo x y "abc
```

This command is not complete, because there is not a matching double-quote for the existing double-quote. Unmatched single-quotes can also result in incomplete commands, as can a variable name with an unmatched "{" character. If `clash` reads a line that is incomplete, then it does not immediately execute that line. Instead, it reads another line (preceded by a "> " prompt, if standard input is a terminal). It continues reading lines until it has one or more complete commands; then it parses the input and executes the commands. If `clash` reaches an end-of-file on its standard input, then it executes whatever input it has, even if it is incomplete; this will result in an error message, if the input was incomplete.

If the standard input for `clash` is not a terminal, `clash` still reads commands from its standard input and processes them as described above. However, in this case `clash` does not output prompts.

Command-Line Arguments

If `clash` is invoked with command-line arguments, then it doesn't read commands from its standard input. If the first command-line argument is "-c", then the second argument is a shell script (one or more commands separated by semi-colons). In this case, `clash` executes that script and then exits.

If the first command-line argument is not "-c", then it must be the name of a file. In this case, `clash` reads commands from that file instead of standard input; `clash` exits when it reaches the end of the file.

Variables and Environment Variables

`Clash` maintains a table of variable bindings, which are used for variable substitutions. Each variable has a string name and a string value. The following variable names have special meanings defined by `clash`:

> `HOME`: the current user's home directory; used for tilde substitutions.
>
> `PATH`: a list of directories in which to search for executables; see below for more information.
>
> `$0`: if `clash` is invoked with no arguments, this is value of its `argv[0]` (the program name under which `clash` was invoked). If `clash` is invoked with the "-c" option, this is the argument just after the argument containing the script, if there is one (if there are no arguments after the script, then $0 holds `argv[0]`. If `clash` is invoked with the name of a file to read commands from, $0 holds that file name.
>
> `$1`, `$2`, etc.: the command-line arguments following $0.
>
> `$#`: the number of command-line arguments after $0.
>
> `$*`: the values of all the command-line arguments after $0, concatenated together and separated by spaces.
>
> `$?`: the exit status returned by the last command; 0 indicates a normal return. In the case of a pipeline, $? reflects the exit status of the last command in the pipeline.

When `clash` starts, it creates one variable for each environment variable received from its parent. When `clash` creates subprocesses, it passes some of its variables to the subprocess as environment variables. These variables are referred to as *exported*. When `clash` creates initial variable bindings from its environment, it marks each of these variables as exported. The `export` built-in command can be used to mark additional variables as exported.

The PATH Variable and its Cache

The PATH variable is used by `clash` to locate executables for commands. It consists of any number of directory names separated by colons. To execute a command, `clash` searches each of the directories in the PATH variable to see if they contain an executable file whose name is the same as the first word of the command. If so, the first matching file that is executable is used (if no executable file is found, the first non-executable one is used).

However, searching through all the directories in the PATH variable is too expensive to perform for each command execution. Instead, you must keep a cache of mappings from command names to executable file names. You must ensure that the cache is updated whenever the PATH variable changes (you do not need to update the cache when files are created or deleted in directories in the path).

If no PATH variable exists, then use "/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin" as the default.

Note: the PATH mechanism is only used if the first word of a command contains no "/" characters. If the first word contains a "/" character, then the word is used directly as the name of the executable, bypassing the PATH mechanism.

Exit Status for Clash

If `clash` encounters an end-of-file condition on its standard input at a point where it has no partially-assembled command, then it exits. It returns the value of the $? variable (the exit status of the most recently executed command) as its exit status, unless the exit is caused by the `exit` built-in command. In that case, the exit status is determined by the argument to `exit`.

Error Handling

There are many error conditions that can occur in `clash`, such as trying to "cd" to a non-existent directory, or the presence of a ">" without a file to redirect to. You must detect these errors and handle them appropriately. In most cases, the right behavior is to abort the current command, print an appropriate message, and continue processing more commands. Try experiments with `bash` to see how it handles various errors, and do the same in `clash`. Try to make your error messages as similar as possible to those from `bash`.

Additional Notes and Requirements

- You must implement the project in C++.
- I will create a private GitHub repository for your team to use and send you information about this repository. Create a branch in this repo named `project3` and use this branch for all of your work on the project.
- The parsing rules for `clash` are pretty complex and intertwined; your challenge for this project is to find a clean way of decomposing the implementation.
- You must write your own parser; you may not use a parser generator or any existing libraries to help with parsing. In general, you should be building from scratch, but you may use any of the C++ std:: classes. If you have any questions about what existing packages it is OK to use, please check with me.
- Here are some functions that you will probably find useful in your implementation:

  `isatty`: determine whether a file descriptor is associated with a console terminal.

  `fork`, `execve`, `waitpid`: create subprocesses and wait for them to complete.

  `opendir`, `readdir`, `closedir`: read the list of file names in a given directory.

  `access`: determine whether a file is executable.

  `getpwnam`: get the password file record for a user, which includes the user's home directory.

  `open`, `dup`, `close`, `pipe`: useful for implementing I/O redirection.
- For at least one non-trivial source file, write the top-level declarations and interface comments before you fill in any of the method bodies. Create a commit on the `project3` branch whose only change is the skeletal version of this file, and tag that commit `commentsBeforeCode3`. Make sure that the commit message also includes the name of this file. I recommend that you write comments before code for *all* your files, but I will only require it for one file.

- As in the past, please use 4-space indents and keep lines to no more than 80 characters in length.
- The repo for this project will contain a file `words.py`. This script will print the elements of its `argv`, so that you can see exactly what is in each word. You may find this useful during testing.
- The repo for this project will also contain a file `test.py`. If you invoke this script, it will test the functionality of your shell, assuming that the executable is in a file named `clash`. Your project should be able to pass all, or almost all, of the tests in this file.

Submitting Your Project

To submit your project, push all of your changes to GitHub (on the `project3` branch) and then create a pull request on GitHub. The base for the pull request should be your `master` branch (which has nothing on it except your initial commit) and the comparison branch should be the head of your `project3` branch. Use "Project 3" as the title for your pull request. If your project is not completely functional at the time you submit, describe what is and isn't working in the comments for the pull request. Include a file `test.out` in your commit, which is the output generated by running `test.py`.

If you are planning to use late days for this project (or any project) please send me an email before the project deadline so that I know how many late days you plan to use.