```c
// smallsh
// By George Hill
// CS 344
// 2020-05-10

// Implements a simple bash-like shell with support for (a) three built-in
// commands (status, cd, and exit), (b) file redirection (with < and >),
// (c) background processes (with &), and (d) otherwise generally calling
// GNU/Linux executables. Ignores Ctrl-C and interprets Ctrl-Z as toggling
// on and off a "foreground-only" mode in which "&" is ignored.

// 80 Columns: ////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>

#define TRUE 1
#define FALSE 0

#define MAX_STRING_LENGTH 2048
#define MAX_COMMAND_ARRAY_SIZE 518
// We will have an array of strings. It will have one string for the command,
// 512 strings for arguments, four strings for redirection symbols and their
// files, and one final string for commands that should be run in the background.
// That adds up to 518 strings.
#define MAX_DIGITS_IN_PROCESS_ID 10 // This is a guess.
#define STATUS_REPORT_MAX_LENGTH 100

#define EXIT_VALUE 1
#define SIGNAL_RECEIVED 0
// The statusType variable in main() will track how the last foreground process
// terminated. If it exited normally, the statusType variable will have the
// value EXIT_VALUE. If it exited because of a signal, it will have the value
// SIGNAL_RECEIVED.

#define COMMAND_AND_ARGUMENT_DELIMITER " " // Must use double-quotes.
// We will use this to separate commands and arguments from each other.

#define PROCESS_NUMBER_SYMBOL '$' // Must use single-quotes because used with
                                  // character comparison.
// We will use this to search for "$$" in the user's input.

#define COMMENT_SYMBOL '#' // Must use single-quotes.
// We will use this to identify comment lines.

#define BACKGROUND_SYMBOL "&" // Must use double-quotes because used with
                              // strcmp().
// We will use this to identify commands that need to run in the background.

#define REDIRECT_INPUT "<" // Must use double-quotes.
#define REDIRECT_OUTPUT ">" // Must use double-quotes.
// We will use these to identify commands that need file redirection.

#define DEV_NULL "/dev/null"
// We will use this with certain background processes.

```

```c
#define EXIT_COMMAND "exit"
#define STATUS_COMMAND "status"
#define CD_COMMAND "cd"
// These are the three built-in commands.

struct runningProcess // Will store process IDs of running proceses in a
                      // linked list.
{
    int processID;
    struct runningProcess* next;
};

int usingBackgroundIsPossible = TRUE;
int receivedSigtstp = FALSE;
int weAreWaitingForForegroundProcessToStop = FALSE;
// As far as I can tell, we need to use global variables so that the shell
// can catch SIGTSTP signal _during execution of a foreground process_ and make
// a plan to act on that signal only after the termination of the foreground
// process. That is to say, I don't know that we have any other way to pass the
// function variables to manipulate.

// The next two functions will be used throughout the rest of the program
// to safely generate output:

void outputStringWithNoNewline(char* text)
{
    printf("%s", text);
    fflush(stdout);
}

void outputStringWithANewline(char* text)
{
    printf("%s\n", text);
    fflush(stdout);
}

// This function toggles our state between the normal mode and the foreground-
// only mode:
void implementSigtstpLogic()
{
    if (usingBackgroundIsPossible == TRUE)
    {
        usingBackgroundIsPossible = FALSE;
        char* message =
            "\nEntering foreground-only mode (& is now ignored)\n";
        write(STDOUT_FILENO, message, 50);
    }
    else
    {
        usingBackgroundIsPossible = TRUE;
        char* message = "\nExiting foreground-only mode\n";
        write(STDOUT_FILENO, message, 30);
    }
}

// This function will be called when our (parent) shell process receives
// a SIGTSTP:
void dealWithSigtstp(int signo)
{
    if (weAreWaitingForForegroundProcessToStop == FALSE)
    {
        // We aren't blocked at waitpid(), so we are (probably) at the
        // command-line prompt, so we should act on SIGTSTP right away.
        implementSigtstpLogic();
```

```
128     }
129     else
130     {
131         // We _are_ blocked at waitpid(), so we hold off on doing anything
132         // about SIGTSTP until the foreground process terminates. However,
133         // in order to do that, we need to update this global variable so
134         // that the work can be done when we are done being blocked at
135         // waitpid().
136         receivedSigtstp = TRUE;
137     }
138     return;
139 }
140 
141 // This function removes the given pid from the linked list that contains
142 // the pid's of background processes. As noted below, writing it was hard.
143 void forget(int processID, struct runningProcess** listOfProcesses)
144 {
145     struct runningProcess* current = *listOfProcesses;
146     // By dereferencing the pointer to a pointer, we end up with "current"
147     // being a pointer that has the same value as "listOfProcesses" in main().
148     // I think.
149     struct runningProcess* previous = NULL;
150 
151     // Iterate over the linked list until locating the link that contains the
152     // pid that we seek:
153     while (current->processID != processID)
154     {
155         previous = current;
156         current = current->next;
157     }
158 
159     if (previous != NULL)
160     {
161         // Remove "current" from the linked list:
162         previous->next = current->next;
163     }
164     else
165     {
166         // "previous" is still NULL, which means that we found the pid in the
167         // very first link, which means that we need to set the head of the
168         // linked list to point to the second element in the list (or to NULL,
169         // which will be the value of "current->next" if there is no second
170         // link).
171         *listOfProcesses = current->next;
172     }
173 
174     free(current);
175 
176     return;
177 }
178 
179 // This function checks to see if the process with the given pid has finished.
180 // If it has, this function calls another function to remove that pid from the
181 // linked list of background processes. As noted below, writing this was
182 // difficult.
183 void checkStatusOfProcess
184 (
185     int processID,
186     struct runningProcess** listOfProcesses
187 )
188 {
189     // printf("checkStatusOfProcess(%d)\n", processID);
190 
191     int exitedOrNot = -5;
```

```
192     int childExitMethod = -5;
193     int statusValue = -5;
194     char statusReport[STATUS_REPORT_MAX_LENGTH];
195 
196     exitedOrNot = waitpid(processID, &childExitMethod, WNOHANG);
197 
198     if (exitedOrNot == 0) {
199         return;
200     }
201 
202     if (WIFEXITED(childExitMethod) != 0)
203     {
204         // The process exited by exit(0), exit(1), return 0, etc.
205         statusValue = WEXITSTATUS(childExitMethod);
206         sprintf
207         (
208             statusReport,
209             "background pid %d is done: exit value %d",
210             processID,
211             statusValue
212         );
213         outputStringWithANewline(statusReport);
214         forget(processID, listOfProcesses);
215     }
216     else if (WIFSIGNALED(childExitMethod) != 0)
217     {
218         // The process exited because of an uncaught signal.
219         statusValue = WTERMSIG(childExitMethod);
220         sprintf
221         (
222             statusReport,
223             "background pid %d is done: terminated by signal %d",
224             processID,
225             statusValue
226         );
227         outputStringWithANewline(statusReport);
228         forget(processID, listOfProcesses);
229     }
230 
231     return;
232 }
233 
234 // This function iterates over a linked list of process IDs from background
235 // processes. For each process ID encountered, it calls a separate function
236 // to see if that process has exited yet. Because this is a linked list using
237 // pointers to pointers, writing it was hard, and so I'm leaving in the
238 // printf() calls used for debugging (although they are commented out).
239 void checkForFinishedBackgroundProcesses
240 (
241     struct runningProcess** listOfProcesses
242 )
243 {
244     struct runningProcess* current = *listOfProcesses;
245     // By dereferencing the pointer to a pointer, we end up with "current"
246     // being a pointer that has the same value as "listOfProcesses" in main().
247     // I think.
248     struct runningProcess* temp = NULL;
249 
250     // printf("checkForFinishedBackgroundProcesses, %p\n", *listOfProcesses);
251 
252     // Iterate over each element in the linked list. If there are no links in
253     // the list, then "current" will equal NULL and we won't look at anything.
254     // If there are links in the list, then "current" will equal NULL when it
255     // finally reaches the last link.
```

```
256     while (current != NULL)
257     {
258         temp = current;
259         current = current->next; // This has to be before the next line because
260                                  // the next line might lead to a forget() call.
261         // printf("current: %p\n", current);
262         checkStatusOfProcess(temp->processID, listOfProcesses);
263     }
264 
265     return;
266 }
267 
268 // Output prompt, get line of user input, and parse each word of that input
269 // into an array, noting the total number of array elements used:
270 void getCommandArray
271 (
272     char commandArray[MAX_COMMAND_ARRAY_SIZE][MAX_STRING_LENGTH],
273     int* arrayElementsUsed
274 )
275 {
276     // Sample code for using getline was provided by the instructor at:
277     // http://web.engr.oregonstate.edu/~brewsteb/CS344Slides/2.4%20File%20Access%20in%20C.pdf
278     // And at:
279     // http://web.engr.oregonstate.edu/~brewsteb/THCodeRepository/userinput_adv.c
280 
281     int numCharsEntered = -5; // Will hold the number of characters entered.
282     size_t bufferSize = 0; // Will hold how large the allocated buffer is.
283     char* lineEntered = NULL; // Will point to a buffer allocated by getline()
284                               // that holds the entered string + \n + \0.
285 
286     while(TRUE)
287     {
288         outputStringWithNoNewline(": "); // Output prompt.
289 
290         numCharsEntered = getline(&lineEntered, &bufferSize, stdin);
291         // Get a line from the user.
292 
293         if (numCharsEntered == -1)
294         {
295             // We got an error, probably because someone sent a SIGTSTP.
296             clearerr(stdin);
297         }
298         else
299         {
300             break;
301         }
302     }
303 
304     lineEntered[numCharsEntered - 1] = 0; // Turn ending \n into a \0.
305 
306     char* token = NULL;
307     int index = 0;
308 
309     token = strtok(lineEntered, COMMAND_AND_ARGUMENT_DELIMITER);
310     while (token != NULL)
311     {
312         strcpy(commandArray[index], token);
313         index++;
314         token = strtok(NULL, COMMAND_AND_ARGUMENT_DELIMITER);
315     }
316 
317     free(lineEntered);
318 
319     *arrayElementsUsed = index;
```

```
320     return;
321 }
322 
323 // Replace each instance of "$$" with the process ID:
324 void replaceDoubleDollarSigns
325 (
326     char commandArray[MAX_COMMAND_ARRAY_SIZE][MAX_STRING_LENGTH],
327     int arrayElementsUsed
328 )
329 {
330     // Iterate over each word in the array:
331     int i;
332     for (i = 0; i < arrayElementsUsed; i++) {
333 
334         int currentWordHasMadeItThrough = FALSE;
335 
336         // The current world will have "made it through" when it can go through
337         // the entire inner iterator without triggering a substitution from
338         // "$$" to "<pid>". It's not enough to run it through once, because
339         // we might have multiple "$$" substrings. After hitting a "$$"
340         // substring, we'll break out of the inner iterator and recheck the
341         // string from the very top.
342         while (currentWordHasMadeItThrough == FALSE)
343         {
344             int wordLength = strlen(commandArray[i]);
345             int weMadeAChange = FALSE;
346 
347             // Iterate over each character in the current word:
348             int j;
349             for (j = 0; j < wordLength - 1; j++)
350             {
351                 char firstChar = commandArray[i][j];
352                 char secondChar = commandArray[i][j + 1];
353 
354                 if (firstChar == PROCESS_NUMBER_SYMBOL &&
355                     secondChar == PROCESS_NUMBER_SYMBOL)
356                 {
357                     // We hit a "$$" substring starting at character j.
358 
359                     char tempWord[MAX_STRING_LENGTH + MAX_DIGITS_IN_PROCESS_ID];
360 
361                     // Copy in the characters that come before the first "$":
362                     int k;
363                     for (k = 0; k < j; k++)
364                     {
365                         tempWord[k] = commandArray[i][k];
366                     }
367 
368                     // Get the process id into a string:
369 
370                     // https://stackoverflow.com/questions/53230155/converting-pid-t-to-string
371 
372                     int pid = getpid();
373                     char tempPidString[MAX_DIGITS_IN_PROCESS_ID];
374                     sprintf(tempPidString, "%d", pid);
375 
376                     // Iterate over the process ID string, copying it in to the
377                     // temporary word:
378                     int tempPidStringLength = strlen(tempPidString);
379                     for (k = 0; k < tempPidStringLength; k++)
380                     {
381                         tempWord[j + k] = tempPidString[k];
382                     }
383 
```

```
384                     // Copy in the characters that come after the "$$"
385                     // substring.
386                     int m = 0;
387                     for (k = j + 2; k < wordLength; k++)
388                     {
389                         tempWord[j + tempPidStringLength + m] =
390                             commandArray[i][k];
391                         m++;
392                     }
393 
394                     // Make sure to mark the end of the temporary string:
395                     tempWord[j + tempPidStringLength + m] = 0;
396 
397                     strcpy(commandArray[i], tempWord);
398 
399                     weMadeAChange = TRUE;
400 
401                     break; // Break out of the for-j loop.
402                 }
403             }
404 
405             if (weMadeAChange == FALSE)
406             {
407                 currentWordHasMadeItThrough = TRUE;
408             }
409         }
410     }
411 
412     return;
413 }
414 
415 // This function helps implemnent the "exit" built-in command. It needs to
416 // terminate any background processes. This function is similar to
417 // checkForFinishedBackgroundProcesses().
418 void prepForExit(struct runningProcess** listOfProcesses)
419 {
420     struct runningProcess* current = *listOfProcesses;
421     // By dereferencing the pointer to a pointer, we end up with "current"
422     // being a pointer that has the same value as "listOfProcesses" in main().
423     // I think.
424     struct runningProcess* temp = NULL;
425 
426     // Iterate over each element in the linked list. If there are no links in
427     // the list, then "current" will equal NULL and we won't look at anything.
428     // If there are links in the list, then "current" will equal NULL when it
429     // finally reaches the last link.
430     while (current != NULL)
431     {
432         temp = current;
433         current = current->next;
434         kill(temp->processID, SIGKILL);
435 
436         // It seems to me that we should output a message noting that the
437         // background process was terminated, so I am including the following:
438 
439         int childExitMethod;
440         int statusValue;
441         char statusReport[STATUS_REPORT_MAX_LENGTH];
442 
443         waitpid(temp->processID, &childExitMethod, 0);
444 
445         if (WIFEXITED(childExitMethod) != 0)
446         {
447             // The process exited by exit(0), exit(1), return 0, etc.
```

```
448             statusValue = WEXITSTATUS(childExitMethod);
449             sprintf
450             (
451                 statusReport,
452                 "background pid %d is done: exit value %d",
453                 temp->processID,
454                 statusValue
455             );
456             outputStringWithANewline(statusReport);
457             forget(temp->processID, listOfProcesses);
458         }
459         else if (WIFSIGNALED(childExitMethod) != 0)
460         {
461             // The process exited because of an uncaught signal.
462             statusValue = WTERMSIG(childExitMethod);
463             sprintf
464             (
465                 statusReport,
466                 "background pid %d is done: terminated by signal %d",
467                 temp->processID,
468                 statusValue
469             );
470             outputStringWithANewline(statusReport);
471             forget(temp->processID, listOfProcesses);
472         }
473     }
474 
475     return;
476 }
477 
478 // This function implements the "status" built-in command:
479 void outputStatus(int statusType, int statusValue)
480 {
481     if (statusType == EXIT_VALUE)
482     {
483         outputStringWithNoNewline("exit value ");
484     }
485     else
486     {
487         outputStringWithNoNewline("terminated by signal ");
488     }
489 
490     char valueOrSignal[5];
491     sprintf(valueOrSignal, "%d", statusValue);
492     outputStringWithANewline(valueOrSignal);
493 
494     return;
495 }
496 
497 // This function implements the "cd" built-in command:
498 void changeDirectory(char parameter[MAX_STRING_LENGTH])
499 {
500     const char* homePath = getenv("HOME");
501     // http://www0.cs.ucl.ac.uk/staff/W.Langdon/getenv/
502 
503     if (strlen(parameter) == 0 || strcmp(parameter, "~") == 0)
504     // Strangely, chdir() wouldn't respond to having a "~" string as
505     // its parameter. It ignored it. In order to make "cd ~" work as
506     // expected, it is necessary to test the parameter for "~" and treat it
507     // as if the user entered just "cd".
508     {
509         chdir(homePath);
510     }
511     else
```

```
512     {
513         chdir(parameter);
514     }
515 
516     return;
517 }
518 
519 // This function adds background-command pids to a linked list. It was
520 // extremely hard to debug, so I'm leaving my debugging printf() statements in
521 // (although they are commented out).
522 void remember(struct runningProcess** listOfProcesses, int processToRemember)
523 {
524     // printf
525     // (
526     //     "remember(%p, %d) and *listOfProcesses = %p\n",
527     //     listOfProcesses,
528     //     processToRemember,
529     //     *listOfProcesses
530     // );
531     if (*listOfProcesses == NULL) {
532 
533         // We don't have any links yet in our linked list, so we have to create
534         // one:
535 
536         // printf("Going to make first link.\n");
537         *listOfProcesses =
538             (struct runningProcess*)malloc(sizeof(struct runningProcess));
539 
540         (*listOfProcesses)->processID = processToRemember;
541         (*listOfProcesses)->next = NULL;
542     } else {
543 
544         // We already have at least one link in our linked list of processes to
545         // remember, so we need to add the current one at the end:
546 
547         // printf("Going to make an additional link.\n");
548         struct runningProcess* current = (*listOfProcesses)->next;
549         struct runningProcess* previous = *listOfProcesses;
550         // printf("ccurent: %p\n", current);
551         while (current != NULL) {
552             previous = current;
553             current = current->next;
554         }
555         // printf("cccurent: %p\n", current);
556 
557         // Having reached the end, we create another link. However, it's
558         // not enough to know "current", because current == NULL, and that
559         // doesn't help us add a link. We need to know "previous" so that we
560         // can add our link to previous->next:
561         previous->next =
562             (struct runningProcess*)malloc(sizeof(struct runningProcess));
563 
564         previous->next->processID = processToRemember;
565         previous->next->next = NULL;
566     }
567 
568     return;
569 }
570 
571 // This function evalutes the command array to see if there is a need for
572 // input/output redirection or running in the background. It then actually
573 // executes the command by using fork() and execvp(). It also deals with the
574 // aftermath of executing a command by waiting for foreground commands (and
575 // noting their manner of termination) and by adding background-command pids
```

```
// to a linked list.
void executeCommand
(
    char commandArray[MAX_COMMAND_ARRAY_SIZE][MAX_STRING_LENGTH],
    int arrayElementsUsed,
    int* statusType,
    int* statusValue,
    int* usingBackgroundIsPossible,
    struct runningProcess** listOfProcesses,
    struct sigaction* originalSigintAction
)
{
    int actuallyRunInBackground = FALSE;

    char fileForInputRedirection[MAX_STRING_LENGTH] = "";
    char fileForOutputRedirection[MAX_STRING_LENGTH] = "";

    // See if there is a BACKGROUND_SYMBOL as the last element in the command
    // array, and if so deal with it:

    if (strcmp(commandArray[arrayElementsUsed - 1], BACKGROUND_SYMBOL) == 0)
    {
        if (*usingBackgroundIsPossible == TRUE)
        {
            actuallyRunInBackground = TRUE;
        }
        arrayElementsUsed--;
    }

    // Check the last two arguments to see if we might be redirecting input or
    // output:

    int needToCheckOneMoreTime = FALSE;

    if (strcmp(commandArray[arrayElementsUsed - 2], REDIRECT_INPUT) == 0)
    {
        strcpy(fileForInputRedirection, commandArray[arrayElementsUsed - 1]);
        arrayElementsUsed = arrayElementsUsed - 2;
        needToCheckOneMoreTime = TRUE;
    }
    else if (strcmp(commandArray[arrayElementsUsed - 2], REDIRECT_OUTPUT) == 0)
    {
        strcpy(fileForOutputRedirection, commandArray[arrayElementsUsed - 1]);
        arrayElementsUsed = arrayElementsUsed - 2;
        needToCheckOneMoreTime = TRUE;
    }

    // If the last two elements indicated redirection, then we also need to
    // check the two elements before them:

    if (needToCheckOneMoreTime == TRUE)
    {
        if (strcmp(commandArray[arrayElementsUsed - 2], REDIRECT_INPUT) == 0)
        {
            strcpy
            (
                fileForInputRedirection,
                commandArray[arrayElementsUsed - 1]
            );
            arrayElementsUsed = arrayElementsUsed - 2;
        }
        else if
        (
            strcmp(commandArray[arrayElementsUsed - 2], REDIRECT_OUTPUT) == 0
```

```
640         )
641         {
642             strcpy
643             (
644                 fileForOutputRedirection,
645                 commandArray[arrayElementsUsed - 1]
646             );
647             arrayElementsUsed = arrayElementsUsed - 2;
648         }
649     }
650 
651     // Now we need to take commandArray and put it in a form that we can send
652     // to execvp():
653 
654     char* commandArgs[arrayElementsUsed];
655 
656     int i;
657     for (i = 0; i < arrayElementsUsed; i++)
658     {
659         commandArgs[i] = calloc(MAX_STRING_LENGTH, sizeof(char));
660         strcpy(commandArgs[i], commandArray[i]);
661     }
662     commandArgs[arrayElementsUsed] = NULL;
663 
664     // NOW WE FORK() AND EXECVP() !!!
665 
666     // Template for forking comes from instructor at:
667     // http://web.engr.oregonstate.edu/~brewsteb/CS344Slides/3.1%20Processes.pdf
668 
669     pid_t spawnPid = -5;
670     int childExitMethod = -5;
671 
672     spawnPid = fork();
673 
674     if (spawnPid == -1) //  Error!
675     {
676         perror("Error when attempting to fork!\n");
677         exit(1);
678     }
679     else if (spawnPid == 0) // We are in the child process!
680     {
681         // If the file is going to run in the background, then we will need to
682         // set up input and output redirection (unless the user has already
683         // specified such redirection):
684         if (actuallyRunInBackground == TRUE)
685         {
686             if (strcmp(fileForInputRedirection, "") == 0)
687             {
688                 strcpy(fileForInputRedirection, DEV_NULL);
689             }
690             if (strcmp(fileForOutputRedirection, "") == 0)
691             {
692                 strcpy(fileForOutputRedirection, DEV_NULL);
693             }
694         }
695 
696         // Now actually set up input redirection, if necessary:
697         if (strcmp(fileForInputRedirection, "") != 0)
698         {
699             // Code for file redirection derived from professor's examples at:
700             // http://web.engr.oregonstate.edu/~brewsteb/CS344Slides/3.4%20More%20UNIX%20IO.pdf
701 
702             int sourceFD = open(fileForInputRedirection, O_RDONLY);
703 
```

```
            if (sourceFD == -1) {
                perror("Error when opening file for input redirection!");
                // printf("cannot open %s for input", fileForInputRedirection);
                exit(1);
            }

            int result = dup2(sourceFD, 0);

            if (result == -1)
            {
                perror("Error when initiating input redirection!");
                exit(1);
            }
        }

        // And actualy set up output redirection, if necessary:
        if (strcmp(fileForOutputRedirection, "") != 0)
        {
            int targetFD = open
            (
                fileForOutputRedirection,
                O_WRONLY | O_CREAT | O_TRUNC,
                0644
            );

            if (targetFD == -1) {
                perror("Error when opening file for output redirection!");
                // printf("cannot open %s for output", fileForOutputRedirection);
                exit(1);
            }

            int result = dup2(targetFD, 1);

            if (result == -1)
            {
                perror("Error when initiating output redirection!");
                exit(1);
            }
        }

        // If the command is going to be run in the _foreground_, we need to
        // set sigaction(SIGINT) back to its original behavior (the behavior
        // it had before we set things to ignore SIGINT):

        if (actuallyRunInBackground == FALSE)
        {
            sigaction(SIGINT, originalSigintAction, NULL);
        }

        // Whether this is going to be a foreground process or a background
        // process--either way--we need to set this child process to ignore
        // SIGTSTP:

        struct sigaction ignoreAction = {{0}};
        ignoreAction.sa_handler = SIG_IGN;
        sigaction(SIGTSTP, &ignoreAction, NULL);

        // And finally we're ready to execvp():

        // Pattern for execvp() comes from instructor at:
        // http://web.engr.oregonstate.edu/~brewsteb/CS344Slides/3.1%20Processes.pdf

        if (execvp(*commandArgs, commandArgs) < 0)
        {
```

```
768             perror("Error when attempting to execute command!");
769             exit(1);
770         }
771     }
772 
773     // Otherwise, we are still in the parent process!
774 
775     if (actuallyRunInBackground == FALSE)
776     {
777         // We're running the command in the foreground, so we have to wait
778         // for it to terminate:
779 
780         // We have to make sure that these globe variables are set correctly
781         // so that we can deal with it if a SIGTSTP comes in while we are
782         // blocked at waitpid().
783         weAreWaitingForForegroundProcessToStop = TRUE;
784         receivedSigtstp = FALSE;
785 
786         int resultPid = -1;
787 
788         while (resultPid == -1)
789         {
790             resultPid = waitpid(spawnPid, &childExitMethod, 0);
791             // If we are blocked here at waitpid() and then receive a
792             // SIGTSTP, waitpid() will return with -1. However, the foreground
793             // process hasn't actually stopped. When that happens, we need to
794             // loop back and waitpid() again until the foreground process
795             // actually stops.
796         }
797 
798         // We should update this global variable.
799         weAreWaitingForForegroundProcessToStop = FALSE;
800 
801         // We deal with it _now_ if a SIGTSTP came in while we were blocked
802         // at waitpid().
803         if (receivedSigtstp == TRUE)
804         {
805             receivedSigtstp = FALSE;
806             implementSigtstpLogic();
807         }
808 
809         // Now we need to update our state variables to reflect the way that
810         // the foreground process terminated:
811 
812         if (WIFEXITED(childExitMethod) != 0)
813         {
814             // The process exited by exit(0), exit(1), return 0, etc.
815             *statusType = EXIT_VALUE;
816             *statusValue = WEXITSTATUS(childExitMethod);
817         }
818         else if (WIFSIGNALED(childExitMethod) != 0)
819         {
820             // The process exited because of an uncaught signal.
821             *statusType = SIGNAL_RECEIVED;
822             *statusValue = WTERMSIG(childExitMethod);
823             printf("terminated by signal %d\n", *statusValue);
824         } else {
825             perror("A process ended for reasons unknown!");
826             exit(1);
827         }
828     } else {
829         // We're running the file in the background, so we aren't going to wait
830         // for it, but we do have to announce that it's in the background:
831 
```

```
832         char backgroundMessage[STATUS_REPORT_MAX_LENGTH];
833         sprintf(backgroundMessage, "background pid is %d", spawnPid);
834         outputStringWithANewline(backgroundMessage);
835 
836         // We also have to add it to our watch list of processes running in the
837         // background:
838 
839         remember(listOfProcesses, spawnPid);
840         // This use of a linked list that involves pointers to pointers almost
841         // blows my mind. It was easy enough to write the linked list part,
842         // but then I realized that passing pointers by value, which I did at
843         // first, wasn't going to let me change what those pointers were
844         // pointing to, so I had to go back and make it use pointers to
845         // pointers.
846     }
847 
848     for (i = 0; i < arrayElementsUsed; i++)
849     {
850         free(commandArgs[i]);
851     }
852 
853     return;
854 }
855 
856 int main()
857 {
858     // The following handful of variables track the program state:
859 
860     char commandArray[MAX_COMMAND_ARRAY_SIZE][MAX_STRING_LENGTH];
861     int arrayElementsUsed = 0;
862 
863     int statusType = EXIT_VALUE;
864     int statusValue = 0;
865 
866     struct runningProcess* listOfProcesses = NULL;
867 
868     // Make shell ignore SIGINT:
869 
870     struct sigaction ignoreAction = {{0}};
871     struct sigaction originalSigintAction = {{0}};
872     // https://stackoverflow.com/questions/13746033/how-to-repair-warning-missing-braces-around-
    initializer
873     ignoreAction.sa_handler = SIG_IGN;
874     sigaction(SIGINT, &ignoreAction, &originalSigintAction);
875 
876     // Make shell handle SIGTSTP:
877 
878     struct sigaction handleSigtstp = {{0}};
879     handleSigtstp.sa_handler = dealWithSigtstp;
880     sigfillset(&handleSigtstp.sa_mask);
881     handleSigtstp.sa_flags = 0; // I don't think this line is necessary.
882     sigaction(SIGTSTP, &handleSigtstp, NULL);
883 
884     // The following is the program's main loop:
885 
886     while (TRUE)
887     {
888         checkForFinishedBackgroundProcesses(&listOfProcesses);
889         // We have to send the _address_ of listOfProcesses, not the value
890         // of the pointer, because we need to be able to change what it's
891         // pointing to in the functions that we're now calling. This almost
892         // blows my mind.
893 
894         getCommandArray(commandArray, &arrayElementsUsed);
```

```
895
896         replaceDoubleDollarSigns(commandArray, arrayElementsUsed);
897
898         // Now we can check to see if we need to invoke one of the three
899         // built-in commands:
900         if (strcmp(commandArray[0], EXIT_COMMAND) == 0)
901         {
902             prepForExit(&listOfProcesses);
903             break;
904         }
905         else if (strcmp(commandArray[0], STATUS_COMMAND) == 0)
906         {
907             outputStatus(statusType, statusValue);
908         }
909         else if (strcmp(commandArray[0], CD_COMMAND) == 0)
910         {
911             if (arrayElementsUsed == 1)
912             {
913                 changeDirectory("");
914             }
915             else
916             {
917                 changeDirectory(commandArray[1]);
918             }
919         }
920         // Or if we're doing nothing:
921         else if (arrayElementsUsed == 0)
922         {
923             // Do nothing; it's a blank line.
924         }
925         else if (commandArray[0][0] == COMMENT_SYMBOL)
926         {
927             // Do nothing; it's a comment line.
928         }
929         // And if none of the above is true, then we need to try to execute
930         // this command by forking and executing:
931         else
932         {
933             executeCommand
934             (
935                 commandArray,
936                 arrayElementsUsed,
937                 &statusType,
938                 &statusValue,
939                 &usingBackgroundIsPossible,
940                 &listOfProcesses,
941                 // We have to send the _address_ of listOfProcesses, not the
942                 // value of the pointer, because we need to be able to change
943                 // what it's pointing to in the functions that we're now
944                 // calling. This almost blows my mind.
945                 &originalSigintAction
946             );
947         }
948     }
949
950     return 0;
951 }
952
```