

./app/main.c

```

1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <string.h>
4: #include <sys/stat.h>
5: #include <unistd.h>
6: #include <errno.h>
7: #include <ctype.h>
8: #include <stdarg.h>
9: #include "colors.h"
10:
11: #define MAX_ARGS 100 // Maximum number of arguments
12: #define MAX_ARG_LEN 256 // Maximum length of each argument
13: #define DEBUG 0 // Change to 1 to verbose debug prints
14:
15: static void
16: die(const char *fmt, ...)
17: {
18:     va_list ap;
19:     va_start(ap, fmt);
20:     vfprintf(stderr, fmt, ap);
21:     va_end(ap);
22:     if (fmt[0] != '\0' && fmt[strlen(fmt) - 1] == ':') {
23:         fputc(' ', stderr);
24:         perror(NULL);
25:     } else {
26:         fputc('\n', stderr);
27:     }
28:     exit(EXIT_FAILURE);
29: }
30:
31: static void *
32: ecalloc(size_t nmemb, size_t size)
33: {
34:     void *p;
35:     if ((p = calloc(nmemb, size)) == NULL)
36:         die("calloc:");
37:     return p;
38: }
39:
40: static void *
41: erealloc(void *p, size_t len)
42: {
43:     if ((p = realloc(p, len)) == NULL)
44:         die("realloc: %s\n", strerror(errno));
45:     return p;
46: }
47:
48: int inArray(const char *str, const char *arr[], int size) {
49:     if (str == NULL) {
50:         return 0;
51:     }
52:     for (int i=0; i<size; i++) {
53:         if (strcmp(str, arr[i]) == 0) {
54:             return 1; // Found
55:         }
56:     }
57:     return 0; // Not Found
58: }
59:
60: char* inPath(const char *command) {
61:
62:     char* path = getenv("PATH");
63:     char *path_copy = strdup(path, strlen(path));
64:
65:     const char split[2] = ":";
66:     struct stat file_stat;
67:
68:     char *token;
69:     token = strtok(path_copy, split);
70:
71:     while (token != NULL) {
72:         char *fullpath = malloc(100);
73:         snprintf(fullpath, 100, "%s/%s", token, command);
74:
75:         if ( (stat(fullpath, &file_stat) == 0) && (file_stat.st_mode & S_IXOTH) ) {
76:             free(path_copy);
77:             return fullpath;
78:         }
79:         token = strtok(NULL, split);
80:         free(fullpath);
81:     }
82:     free(path_copy);
83:     return NULL;
84: }
85:
86: void trim(char *s) {
87:     // Two pointers initially at the beginning
88:     int i = 0, j = 0;
89:
90:     // Skip leading spaces
91:     while (s[i] == ' ') i++;
92:
93:     // Shift the characters of string to remove
94:     // leading spaces
95:     while (s[j++] = s[i++]);
96: }
97:
98: int tokenize(char **tokens, char *str) {
99:     char separator[2] = " ";
100:     char* next_token;
101:     tokens[0] = strtok(str, separator);
102:     next_token = strtok(0, separator);
103:     int i=1;
104:     while (next_token != 0) {
105:         tokens[i] = malloc(strlen(next_token)+1);
106:         strcpy(tokens[i], next_token);
107:         next_token = strtok(0, separator);
108:         i++;
109:     }
110:     return i;
111: }
112:
113: static char *util_cat(char *dest, char *end, const char *str)
114: {
115:     while (dest < end && *str)
116:         *dest++ = *str++;
117:     return dest;
118: }
119:
120: size_t join_str(char *out_string, size_t out_bufsz, const char *delim, char **chararr)
121: {
122:     char *ptr = out_string;
123:     char *strend = out_string + out_bufsz;
124:     while (ptr < strend && *chararr)
125:     {
126:         ptr = util_cat(ptr, strend, *chararr);
127:     }
128: }

```

```

126:         chararr++;
127:         if (*chararr)
128:             ptr = util_cat(ptr, strend, delim);
129:     }
130:     return ptr - out_string;
131: }
132: void parse_line(const char *line, char *command_output, char **args_output, int *arg
_count) {
133:     const char *ptr = line; // Pointer to traverse the input string
134:     char *arg = NULL;
135:     int index = 0;
136:
137:     // Skip leading spaces
138:     while (*ptr && isspace((unsigned char)*ptr)) {
139:         ptr++;
140:     }
141:
142:     // Extract the command (first token)
143:     while (*ptr && !isspace((unsigned char)*ptr)) {
144:         *command_output++ = *ptr++;
145:     }
146:     *command_output = '\0'; // Null-terminate the command string
147:
148:     // Parse the arguments
149:     while (*ptr) {
150:         // Skip leading spaces
151:         while (*ptr && isspace((unsigned char)*ptr)) {
152:             ptr++;
153:         }
154:
155:         if (*ptr == '\0') {
156:             break; // End of string
157:         }
158:
159:         if (*ptr == '\\') { // Handle quoted argument
160:             ptr++; // Skip the opening single quote
161:             arg = malloc(MAX_ARG_LEN);
162:             int arg_pos = 0;
163:
164:             while (*ptr && *ptr != '\\') {
165:                 if (arg_pos < MAX_ARG_LEN - 1) {
166:                     arg[arg_pos++] = *ptr++;
167:                 } else {
168:                     fprintf(stderr, "Argument exceeds maximum length\n");
169:                     free(arg);
170:                     return;
171:                 }
172:             }
173:
174:             if (*ptr == '\\') {
175:                 ptr++; // Skip the closing single quote
176:             }
177:             arg[arg_pos] = '\0';
178:         } else { // Handle unquoted argument
179:             arg = malloc(MAX_ARG_LEN);
180:             int arg_pos = 0;
181:
182:             while (*ptr && !isspace((unsigned char)*ptr)) {
183:                 if (arg_pos < MAX_ARG_LEN - 1) {
184:                     arg[arg_pos++] = *ptr++;
185:                 } else {
186:                     fprintf(stderr, "Argument exceeds maximum length\n");
187:                     free(arg);

```

```

188:                 return;
189:             }
190:         }
191:         arg[arg_pos] = '\0';
192:     }
193:
194:     // Add the argument to the output array
195:     if (index < MAX_ARGS) {
196:         args_output[index++] = arg;
197:     } else {
198:         fprintf(stderr, "Too many arguments\n");
199:         free(arg);
200:         return;
201:     }
202: }
203:
204: *arg_count = index; // Update the argument count
205: }
206:
207:
208: int main() {
209:     // Flush after every printf
210:     setbuf(stdout, NULL);
211:
212:     const char *commands[] = {"exit", "echo", "type", "pwd", "cd"};
213:     int commands_size = sizeof(commands) / sizeof(commands[0]);
214:
215:     char exit_command[]="exit";
216:     char echo_command[]="echo";
217:     char type_command[]="type";
218:     char pwd_command[]="pwd";
219:     char cd_command[]="cd";
220:
221:     char s[2] = " ";
222:     char quote[2] = "\"";
223:
224:     int running = 1;
225:     while (running) {
226:         printf(BHGRN "$ " CRESET);
227:
228:         // Wait for user input
229:         char input[100];
230:         printf(CYN);
231:         fgets(input, 100, stdin);
232:         printf(CRESET);
233:
234:         input[strcspn(input, "\n")] = '\0';
235:
236:         char **argv = (char**)malloc(5*sizeof(char*));
237:         char *args = calloc(8, 128);
238:         char *args_quotes = calloc(8, 128);
239:         int argc = 0;
240:         char cmd[256];
241:
242:         parse_line(input, cmd, argv, &argc);
243:
244:         if (argc > 0) {
245:             join_str(args, 1024, s, argv);
246:
247:             for (int i=0; i<argc; i++){
248:                 strcat(args_quotes, "");
249:                 strcat(args_quotes, argv[i] );
250:                 strcat(args_quotes, "");

```

./app/main.c

```
251:         strcat(args_quotes, " ");
252:     }
253: }
254: else {
255:     args = NULL;
256:     args_quotes = NULL;
257: }
258:
259: if (DEBUG) {
260:     for (int i=0; i<argc; i++) {
261:         printf("argv %d: %s\n", i, argv[i]);
262:     }
263:     printf("cmd: %s\nargc: %d\nargs: %s\nargsq: %s\n---\n", cmd, argc, args, args_q
notes);
264: }
265: if (cmd == NULL) { continue; }
266:
267: // EXIT COMMAND
268: if (strcmp(cmd, exit_command) == 0) {
269:     running=0;
270:     break;
271: }
272:
273: // ECHO COMMAND
274: else if (strcmp(cmd, echo_command) == 0) {
275:     if (args != NULL) {
276:         printf("%s\n", args);
277:     }
278:
279: // TYPE COMMAND
280: else if (strcmp(cmd, type_command) == 0) {
281:     if (argc == 0) {printf("You need to specify a command\n");
282:     }
283:     else {
284:
285:         if (inArray(argv[0], commands, commands_size) == 1) {
286:             printf("%s is a shell builtin\n", argv[0]);
287:         }
288:
289:         else if (inPath(argv[0]) != NULL) {
290:             char* fp;
291:             fp = inPath(argv[0]);
292:             printf("%s is %s\n", argv[0], fp);
293:             free(fp);
294:         }
295:         else { printf("%s: not found\n", argv[0]);}
296:     }
297:
298: // PWD COMMAND
299: else if (strcmp(cmd, pwd_command) == 0) {
300:     char cwd[1024];
301:     getcwd(cwd, sizeof(cwd));
302:     printf("%s\n", cwd);
303: }
304:
305: else if (strcmp(cmd, cd_command) == 0) {
306:     if (argc == 0) {printf("You need to specify a directory\n");
307:     }
308:     else {
309:         if (!strcmp(argv[0], "~")) {
310:             strcpy(args, getenv("HOME"));
311:         }
312:
```

```
313:         int result = chdir(args);
314:         if ((result != 0) && (errno == ENOENT)) {
315:             printf("cd: %s: No such file or directory\n", args);
316:         }
317:     }
318: }
319:
320: // EXECUTE COMMAND IN PATH
321: else if ( inPath(cmd) != NULL ) {
322:
323:     char* fullCommand = calloc(1, 100);
324:     if (args_quotes != NULL) {
325:         snprintf(fullCommand, 100, "%s %s\n", cmd, args_quotes);
326:     }
327:     else {
328:         snprintf(fullCommand, 100, "%s\n", cmd);
329:     }
330:     int returnCode = system(fullCommand);
331:     free(fullCommand);
332: }
333:
334: else {
335:     printf("%s: command not found\n", input);
336: }
337:
338: free(argv);
339: free(args);
340: }
341: return 0;
342: }
```