```c
#include "builtin.h"

// returns true if the 'exit' call
// should be performed
int exit_shell(struct execcmd *parsed) {
    return strncmp(parsed->scmd, "exit", 4) == 0;
}

// returns true if "chdir" was performed
// this means that if 'cmd' contains:
//      $ cd directory (change to 'directory')
//      $ cd (change to HOME)
// it has to be executed and then return true
int cd(struct execcmd *parsed) {
    bool invoked = false;
    if (parsed->argc > 0 && strcmp(parsed->argv[0], "cd") == 0) {
        if (chdir(parsed->argc == 1 ? getenv("HOME") : parsed->argv[1]) == -1)
{
            perror(SHELL_NAME);
        } else {
            char *current_dir = get_current_dir_name();
            snprintf(prompt, sizeof prompt, "(%s)", current_dir);
            free(current_dir);
        }
        invoked = true;
    }
    return invoked;
}

// returns true if 'pwd' was invoked
// in the command line
int pwd(struct execcmd *parsed) {
    bool invoked = false;
    if (parsed->argc > 0 && strcmp(parsed->argv[0], "pwd") == 0) {
        char *current_dir = get_current_dir_name();
        printf("%s\n", current_dir);
        free(current_dir);
        invoked = true;
    }
    return invoked;
}
```

```c
#ifndef BUILTIN_H
#define BUILTIN_H

#include "parsing.h"
#include "defs.h"

extern char prompt[PRMTLEN];

int cd(struct execcmd *parsed);

int exit_shell(struct execcmd *parsed);

int pwd(struct execcmd *parsed);

#endif // BUILTIN_H
```

```c
#include "createcmd.h"

// creates an execcmd struct to store
// the args and environ vars of the command
struct cmd *exec_cmd_create(char *buf_cmd) {

    struct execcmd *e;

    e = (struct execcmd *) calloc(sizeof(*e), sizeof(*e));

    e->type = EXEC;
    strcpy(e->scmd, buf_cmd);

    return (struct cmd *) e;
}

// creates a backcmd struct to store the
// background command to be executed
struct cmd *back_cmd_create(struct cmd *c) {

    struct backcmd *b;

    b = (struct backcmd *) calloc(sizeof(*b), sizeof(*b));

    b->type = BACK;
    strcpy(b->scmd, c->scmd);
    b->c = c;

    return (struct cmd *) b;
}

// encapsulates two commands into one pipe struct
struct cmd *pipe_cmd_create(struct cmd *left, struct cmd *right) {

    if (!right)
        return left;

    struct pipecmd *p;

    p = (struct pipecmd *) calloc(sizeof(*p), sizeof(*p));

    p->type = PIPE;
    p->leftcmd = left;
    p->rightcmd = right;

    return (struct cmd *) p;
}
```

```c
#ifndef CREATECMD_H
#define CREATECMD_H

#include "defs.h"
#include "types.h"

struct cmd *exec_cmd_create(char *cmd);

struct cmd *back_cmd_create(struct cmd *c);

struct cmd *pipe_cmd_create(struct cmd *l, struct cmd *r);

#endif // CREATECMD_H
```

```c
#ifndef DEFS_H
#define DEFS_H

#define _GNU_SOURCE

#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>

#define SHELL_NAME "shell"
#define UNUSED(x) (void)(x)

// color scape strings
#define COLOR_BLUE "\x1b[34m"
#define COLOR_RED "\x1b[31m"
#define COLOR_RESET "\x1b[0m"

#define END_STRING '\0'
#define END_LINE '\n'
#define SPACE ' '

#define BUFLEN 1024
#define PRMTLEN 1024
#define MAXARGS 20
#define ARGSIZE 1024
#define FNAMESIZE 200

// Command representation after parsed
#define EXEC 1
#define BACK 2
#define REDIR 3
#define PIPE 4

// Fd for pipes
#define READ 0
#define WRITE 1

#define EXIT_SHELL 1

#endif //DEFS_H
```

```c
#include "exec.h"

int replace_fd(int newfd, int oldfd) {
    if (dup2(oldfd, newfd) == -1) {
        perror(SHELL_NAME);
        close(oldfd);
        return 1;
    }
    return 0;
}

int replace(int newfd, char *file_name, int flags) {
    int oldfd;
    if (file_name[0] == '&') {
        oldfd = (int) strtol(file_name + 1, NULL, 10);
        if (oldfd == 0) {
            return replace(STDOUT_FILENO, file_name + 1, flags)
                   + replace_fd(STDERR_FILENO, STDOUT_FILENO);
        }
    } else {
        if (file_name[0] == '>') {
            file_name += 1;
            flags |= O_APPEND;
        }
        oldfd = open(file_name, flags,
                     S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
        if (oldfd == -1) {
            char buf[BUFLEN] = {0};
            snprintf(buf, sizeof buf, "%s: %s", SHELL_NAME, file_name);
            perror(buf);
            return 1;
        }
    }
    return replace_fd(newfd, oldfd);
}

// executes a command - does not return
//
// Hint:
// - check how the 'cmd' structs are defined
//      in types.h
void exec_cmd(struct cmd *cmd) {
    struct execcmd *execcmd = (struct execcmd *) cmd;
    struct backcmd *backcmd = (struct backcmd *) cmd;
    struct pipecmd *pipecmd = (struct pipecmd *) cmd;
    pid_t p;
    switch (cmd->type) {
        case EXEC:
            // spawns a command
            for (int i = 0; execcmd->eargv[i] != NULL; i++) {
                char *name = execcmd->eargv[i];
                char *value = NULL;
                int position = 0;
                while (value == NULL) {
                    if (name[position] == '=') {
                        name[position] = 0;
                        value = name + position + 1;
                    } else {
                        position++;
                    }
                }
                if (setenv(name, value, true) == -1) {
                    char buf[BUFLEN] = {0};
                    snprintf(buf, sizeof buf,
                             "cannot define environment variable %s", name);
                    perror(buf);
                }
                name[position] = '=';
            }
            if (execvp(
                    execcmd->argv[0],
                    execcmd->argv
            ) == -1) {
                perror(SHELL_NAME);
            }
            break;
        case BACK: {
            // runs a command in background
            exec_cmd(backcmd->c);
            break;
        }
        case REDIR: {
            // changes the input/output/stderr flow
            int errors = 0;
            if (execcmd->in_file[0] != 0) {
                errors += replace(STDIN_FILENO, execcmd->in_file,
                                  O_RDONLY);
            }
            if (execcmd->out_file[0] != 0)
                errors += replace(STDOUT_FILENO, execcmd->out_file,
                                  O_CREAT | O_WRONLY);
            if (execcmd->err_file[0] != 0)
                errors += replace(STDERR_FILENO, execcmd->err_file,
                                  O_CREAT | O_WRONLY);
            if (errors > 0) return;
            cmd->type = EXEC;
            exec_cmd(cmd);
            break;
        }
        case PIPE: {
            // pipes two commands
            int pipefd[2];
            if (pipe(pipefd) == -1) { perror(SHELL_NAME); }

            if ((p = fork()) == 0) {
                close(pipefd[1]); // Close unused write end
                replace_fd(STDIN_FILENO, pipefd[0]);
                exec_cmd(pipecmd->rightcmd);
            } else if (p > 0) {
                close(pipefd[0]); // Close unused read end
                replace_fd(STDOUT_FILENO, pipefd[1]);
                exec_cmd(pipecmd->leftcmd);
            } else {
                perror(SHELL_NAME);
            }
            break;
        }
        default:
            break;
    }
}
```

```c
#ifndef EXEC_H
#define EXEC_H

#include "defs.h"
#include "types.h"
#include "utils.h"
#include "freecmd.h"

extern struct cmd *parsed_pipe;

void exec_cmd(struct cmd *c);

#endif // EXEC_H
```

```
#include "freecmd.h"

// frees the memory allocated
// for the tree structure command
void free_command(struct cmd *cmd) {

    int i;
    struct pipecmd *p;
    struct execcmd *e;
    struct backcmd *b;

    if (cmd->type == PIPE) {

        p = (struct pipecmd *) cmd;

        free_command(p->leftcmd);
        free_command(p->rightcmd);

        free(p);
        return;
    }

    if (cmd->type == BACK) {

        b = (struct backcmd *) cmd;

        free_command(b->c);
        free(b);
        return;
    }

    e = (struct execcmd *) cmd;

    for (i = 0; i < e->argc; i++)
        free(e->argv[i]);

    for (i = 0; i < e->eargc; i++)
        free(e->eargv[i]);

    free(e);
}
```

```
#ifndef FREECMD_H
#define FREECMD_H

#include "defs.h"
#include "types.h"

void free_command(struct cmd *c);

#endif // FREECMD_H
```

```c
#include <bits/sigset.h>
#include "defs.h"
#include "readline.h"
#include "runcmd.h"

char prompt[PRMTLEN] = {0};
char preprompt[BUFLEN] = {0};
char last_back[BUFLEN] = {0};
pid_t last_back_pid = 0;

// runs a shell command
static void run_shell() {

    char *cmd;

    while ((cmd = read_line(prompt)) != NULL)
        if (run_cmd(cmd) == EXIT_SHELL)
            return;
}

// initialize the shell
// with the "HOME" directory
static void init_shell() {

    char buf[BUFLEN] = {0};
    char *home = getenv("HOME");

    if (chdir(home) < 0) {
        snprintf(buf, sizeof buf, "cannot cd to %s ", home);
        perror(buf);
    } else {
        snprintf(prompt, sizeof prompt, "(%s)", home);
    }
}

void listen_to_your_dying_child(int signal, siginfo_t *siginfo, void *context) {
    UNUSED(signal);
    UNUSED(context);
    if (siginfo->si_pid == last_back_pid) {
        snprintf(preprompt, PRMTLEN,
                 "==> terminado: PID=%d (%s)", siginfo->si_pid, last_back);
        preprompt[PRMTLEN - 1] = 0;
    }
}

void pay_attention_to_your_children() {
    struct sigaction action;
    sigset_t set;
    sigemptyset(&set);
    action.sa_mask = set;
    action.sa_flags = SA_SIGINFO | SA_RESTART;
    action.sa_sigaction = &listen_to_your_dying_child;
    sigaction(SIGCHLD, &action, NULL);
}

int main(void) {
    pay_attention_to_your_children();
    init_shell();
    run_shell();
    return 0;
}
```

```c
#include "parsing.h"
#include "printstatus.h"

// parses an argument of the command stream input
static char *get_token(char *buf, int idx) {

    char *tok;
    int i;

    tok = (char *) calloc(ARGSIZE, sizeof(char));
    i = 0;

    while (buf[idx] != SPACE && buf[idx] != END_STRING) {
        tok[i] = buf[idx];
        i++;
        idx++;
    }

    return tok;
}

// parses and changes stdin/out/err if needed
static bool parse_redir_flow(struct execcmd *c, char *arg) {

    int inIdx, outIdx;

    // flow redirection for output
    if ((outIdx = block_contains(arg, '>')) >= 0) {
        switch (outIdx) {
            // stdout redir
            case 0: {
                strcpy(c->out_file, arg + 1);
                break;
            }
                // stderr redir
            case 1: {
                strcpy(c->err_file, &arg[outIdx + 1]);
                break;
            }
        }

        free(arg);
        c->type = REDIR;

        return true;
    }

    // flow redirection for input
    if ((inIdx = block_contains(arg, '<')) >= 0) {
        // stdin redir
        strcpy(c->in_file, arg + 1);

        c->type = REDIR;
        free(arg);

        return true;
    }

    return false;
}

// parses and sets a pair KEY=VALUE
// environment variable
static bool parse_environ_var(struct execcmd *c, char *arg) {

    // sets environment variables apart from the
    // ones defined in the global variable "environ"
    if (block_contains(arg, '=') > 0) {
        // checks if the KEY part of the pair
        // does not contain a '-' char which means
        // that it is not a environ var, but also
        // an argument of the program to be executed
        // (For example:
        //      ./prog -arg=value
        //      ./prog --arg=value
        // )
        if (block_contains(arg, '-') < 0) {
            c->eargv[c->eargc++] = arg;
            return true;
        }
    }

    return false;
}

// this function will be called for every token, and it should
// expand environment variables. In other words, if the token
// happens to start with '$', the correct substitution with the
// environment value should be performed. Otherwise the same
// token is returned.
//
// Hints:
// - check if the first byte of the argument
//      contains the '$'
// - expand it and copy the value
//       to 'arg'
static char *expand_environ_var(char *arg) {
    if (arg[0] == '$') {
        char *env;
        if (arg[1] == '?' && arg[2] == 0) {
            snprintf(arg, ARGSIZE, "%d", status);
        } else if ((env = getenv(&arg[1])) != NULL) {
            strncpy(arg, env, ARGSIZE);
            arg[ARGSIZE - 1] = 0;
        }
    }
    return arg;
}

// parses one single command having into account:
// - the arguments passed to the program
// - stdin/stdout/stderr flow changes
// - environment variables (expand and set)
static struct cmd *parse_exec(char *buf_cmd) {

    struct execcmd *c;
    char *tok;
    int idx = 0, argc = 0;

    c = (struct execcmd *) exec_cmd_create(buf_cmd);

    while (buf_cmd[idx] != END_STRING) {

        tok = get_token(buf_cmd, idx);
        idx = idx + strlen(tok);

        if (buf_cmd[idx] != END_STRING)
            idx++;

        if (parse_redir_flow(c, tok))
            continue;

        if (parse_environ_var(c, tok))
            continue;

        tok = expand_environ_var(tok);
```

```
        c->argv[argc++] = tok;
    }

    c->argv[argc] = (char *) NULL;
    c->argc = argc;

    return (struct cmd *) c;
}

// parses a command knowing that it contains
// the '&' char
static struct cmd *parse_back(char *buf_cmd) {

    int i = 0;
    struct cmd *e;

    while (buf_cmd[i] != '&')
        i++;

    buf_cmd[i] = END_STRING;

    e = parse_exec(buf_cmd);

    return back_cmd_create(e);
}

// parses a command and checks if it contains
// the '&' (background process) character
static struct cmd *parse_cmd(char *buf_cmd) {

    if (strlen(buf_cmd) == 0)
        return NULL;

    int idx;

    // swaps &> with >&
    if ((idx = block_contains(buf_cmd, '>')) >= 0 &&
        buf_cmd[idx - 1] == '&') {
        buf_cmd[idx] = '&';
        buf_cmd[idx - 1] = '>';
    }

    // checks if the background symbol (&) is after
    // a redir symbol (>), in which case
    // it does not have to run in in the 'back'
    if ((idx = block_contains(buf_cmd, '&')) >= 0 &&
        buf_cmd[idx - 1] != '>')
        return parse_back(buf_cmd);

    return parse_exec(buf_cmd);
}

// parses the command line
// looking for the pipe character '|'
struct cmd *parse_line(char *buf) {

    struct cmd *r, *l;

    char *right = split_line(buf, '|');

    l = parse_cmd(buf);
    r = right[0] != 0 ? parse_line(right) : NULL;

    return pipe_cmd_create(l, r);
}
```

```c
#ifndef PARSING_H
#define PARSING_H

#include "defs.h"
#include "types.h"
#include "createcmd.h"
#include "utils.h"

struct cmd *parse_line(char *b);

#endif // PARSING_H
```

```c
#include "printstatus.h"

// prints information of process' status
void print_status_info(struct cmd *cmd) {

    if (strlen(cmd->scmd) == 0
        || cmd->type == PIPE)
        return;

    if (WIFEXITED(status)) {

        fprintf(stdout, "%s        Program: [%s] exited, status: %d %s\n",
                COLOR_BLUE, cmd->scmd, WEXITSTATUS(status), COLOR_RESET);
        status = WEXITSTATUS(status);

    } else if (WIFSIGNALED(status)) {

        fprintf(stdout, "%s        Program: [%s] killed, status: %d %s\n",
                COLOR_BLUE, cmd->scmd, -WTERMSIG(status), COLOR_RESET);
        status = -WTERMSIG(status);

    } else if (WTERMSIG(status)) {

        fprintf(stdout, "%s        Program: [%s] stopped, status: %d %s\n",
                COLOR_BLUE, cmd->scmd, -WSTOPSIG(status), COLOR_RESET);
        status = -WSTOPSIG(status);
    }
}

// prints info when a background process is spawned
void print_back_info(struct cmd *back) {

    fprintf(stdout, "%s [PID=%d] %s\n",
            COLOR_BLUE, back->pid, COLOR_RESET);
}
```

```
#ifndef PRINTSTATUS_H
#define PRINTSTATUS_H

#include "defs.h"
#include "types.h"

extern int status;

void print_status_info(struct cmd *cmd);

void print_back_info(struct cmd *back);

#endif // PRINTSTATUS_H
```

```
#include "defs.h"
#include "readline.h"

static char buffer[BUFLEN];

// read a line from the standard input
// and prints the prompt
char *read_line(const char *prompt) {

    int i = 0, c = 0;

    if (preprompt[0] != 0) {
        fprintf(stdout, "%s %s %s\n", COLOR_RED, preprompt, COLOR_RESET);
        preprompt[0] = 0;
    }
    fprintf(stdout, "%s %s %s\n", COLOR_RED, prompt, COLOR_RESET);
    fprintf(stdout, "%s", "$ ");

    memset(buffer, 0, BUFLEN);

    c = getchar();

    while (c != END_LINE && c != EOF) {
        buffer[i++] = c;
        c = getchar();
    }

    // if the user press ctrl+D
    // just exit normally
    if (c == EOF)
        return NULL;

    buffer[i] = END_STRING;

    return buffer;
}
```

```c
#ifndef READLINE_H
#define READLINE_H

extern char preprompt[BUFLEN];

char *read_line(const char *prompt);

#endif //READLINE_H
```

```c
#include "runcmd.h"

int status = 0;
struct cmd *parsed_pipe;

// runs the command in 'cmd'
int run_cmd(char *cmd) {

    pid_t p;
    // parses the command line
    struct cmd *parsed = parse_line(cmd);

    // if the "enter" key is pressed
    // just print the prompt again
    if (cmd[0] == END_STRING) {
        free_command(parsed);
        return 0;
    }

    if (parsed->type == EXEC) {
        // cd built-in call
        if (cd((struct execcmd *) parsed)) {
            free_command(parsed);
            return 0;
        }

        // exit built-in call
        if (exit_shell((struct execcmd *) parsed)) {
            free_command(parsed);
            return EXIT_SHELL;
        }

        // pwd built-in call
        if (pwd((struct execcmd *) parsed)) {
            free_command(parsed);
            return 0;
        }
    }

    // forks and run the command
    if ((p = fork()) == 0) {

        // keep a reference
        // to the parsed pipe cmd
        // so it can be freed later
        if (parsed->type == PIPE)
            parsed_pipe = parsed;

        exec_cmd(parsed);
    }

    // store the pid of the process
    parsed->pid = p;

    // background process special treatment
    if (parsed->type != BACK) {
        // waits for the process to finish
        waitpid(p, &status, 0);
        print_status_info(parsed);
    } else {
        strncpy(last_back, ((struct backcmd *) parsed)->c->scmd, BUFLEN);
        last_back_pid = p;
        print_back_info(parsed);
    }

    free_command(parsed);

    return 0;
}
```

```c
#ifndef RUNCMD_H
#define RUNCMD_H

#include "defs.h"
#include "parsing.h"
#include "exec.h"
#include "printstatus.h"
#include "freecmd.h"
#include "builtin.h"

int run_cmd(char *cmd);

extern char last_back[BUFLEN];

extern int last_back_pid;

#endif // RUNCMD_H
```

```c
#ifndef TYPES_H
#define TYPES_H

/* Commands definition types */

/*
 cmd: Generic interface
        that represents a single command.
        All the other *cmd structs can be
        casted to it, and they donÂ´t lose
        information (for example the 'type' field).

        - type: {EXEC, REDIR, BACK, PIPE}
        - pid: the process id
        - scmd: a string representing the command before being parsed
*/
struct cmd {
    int type;
    pid_t pid;
    char scmd[BUFLEN];
};

/*
  execcmd: It contains all the relevant
        information to execute a command.

        - type: could be EXEC or REDIR
        - argc: arguments quantity after parsed
        - eargc: environ vars quantity after parsed
        - argv: array of strings representig the arguments
            of the form: {"binary/command", "arg0", "arg1", ... , (char*)NULL}
        - eargv: array of strings of the form: "KEY=VALUE"
                representing the environ vars
        - *_file: string that contains the name of the file
                to be redirected to

        IMPORTANT: an execcmd struct can have EXEC or REDIR type
                depending on if the command to be executed
                has at least one redirection symbol (<, >, >>, >&)
*/
struct execcmd {
    int type;
    pid_t pid;
    char scmd[BUFLEN];
    int argc;
    int eargc;
    char *argv[MAXARGS];
    char *eargv[MAXARGS];
    char out_file[FNAMESIZE];
    char in_file[FNAMESIZE];
    char err_file[FNAMESIZE];
};

/*
  pipecmd: It contains the same information as 'cmd'
        plus two fields representing the left and right part
        of a command of the form: "command1 arg1 arg2 | command2 arg3"
        As they are of type 'struct cmd',
        it means that they can be either an EXEC or a REDIR command.
*/
struct pipecmd {
    int type;
    pid_t pid;
    char scmd[BUFLEN];
    struct cmd *leftcmd;
    struct cmd *rightcmd;
};

/*
```

```c
  backcmd: It contains the same information as 'cmd'
        plus one more field containing the command to be executed.
        Take a look to the parsing.c file to understand it better.
        Again, this extra field, can have type either EXEC or REDIR
        depending on if the process to be executed in the background
        contains redirection symbols.
*/
struct backcmd {
    int type;
    pid_t pid;
    char scmd[BUFLEN];
    struct cmd *c;
};

#endif // TYPES_H
```

```c
#include "utils.h"

// splits a string line in two
// acording to the splitter character
char *split_line(char *buf, char splitter) {

    int i = 0;

    while (buf[i] != splitter &&
           buf[i] != END_STRING)
        i++;

    buf[i++] = END_STRING;

    while (buf[i] == SPACE)
        i++;

    return &buf[i];
}

// looks in a block for the 'c' character
// and returns the index in which it is, or -1
// in other case
int block_contains(char *buf, char c) {

    for (int i = 0; i < (int) strlen(buf); i++)
        if (buf[i] == c)
            return i;

    return -1;
}
```

```c
#ifndef UTILS_H
#define UTILS_H

#include "defs.h"

char *split_line(char *buf, char splitter);

int block_contains(char *buf, char c);

#endif // UTILS_H
```

Table of Contents