```c
/**
 * Janet Davis, May 5, 2010
 * This program reads a file containing a list of commands, parses each
 * command, and executes each command in a new process.  It then waits for
 * all commands to terminate.
 *
 * Acknowledgments: This program is based on the discussion in Gary Nutt's
 * _Operating Systems_, 3rd edition, pp. 61-65.
 *
 * Jerod Weinman, 10 August 2012
 * Added detailed error messages and strsep over obsolescent strtok.
 **/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>
#include <assert.h>
#include <errno.h>
#include "launch_tests.h"

#define MAX_LINE_LEN 256
#define MAX_ARGS (MAX_LINE_LEN/2)
#define WHITESPACE " \t\n"

typedef struct {                /* Struct to contain a parsed command */
    char* name;
    int argc;
    char* argv[MAX_ARGS+1];
} command_t;

/* Given a command line and a pointer to an empty command_t structure,
 * this function will parse the command line and set values for name,
 * argc, and argv.
 * Produces:
 *  return, an int
 * Preconditions:
 * * cmdline is null-terminated.
 * * strlen(cmdline) < MAX_LINE_LENGTH
 * * cmd points to a valid command_t struct
 * Postconditions:
 * * cmdline is modified by strsep (null-terminators may be inserted)
 * * 0 <= cmd->argc <= MAX_ARGS
 * * cmd->argc is the number of "tokens" or words on the command line
 * * cmd->argv[cmd->argc] is a null pointer
 * * cmd->argv[0] through cmd->argv[cmd->argc-1] are pointers to those tokens
 *     (which are copied from cmdline)
 * * name = cmd->argv[0]
 * * return == EXIT_SUCCESS when all postconditions are met, otherwise
 *   return == EXIT_FAILURE
 */
int parse_command(char *cmdline, command_t *cmd)
{
    int argc = 0;
    char* word;

    /* Fill argv. */
    word = strsep(&cmdline, WHITESPACE);                    /* Get the first token
*/
    while ( word ) {                                        /* Any more tokens? */
      if (strlen(word)) {                                   /* If non-empty token *
/
        cmd->argv[argc] = (char *) malloc(strlen(word)+1);  /* Make space for word
*/

        if  (NULL == cmd->argv[argc]) {                     /* Verify malloc */
          fprintf(stderr,
                  "launch: Error allocating memory for argument \"%s\": %s\n",
                  word, strerror(errno));
          return EXIT_FAILURE;
        }

        strcpy(cmd->argv[argc], word);                      /* Copy word to struct
*/
        argc++;
      }
      word = strsep(&cmdline, WHITESPACE);                  /* Get next token */
    }

    cmd->argv[argc] = NULL;

    /* Set argc and the command name. */
    cmd->argc = argc;
    cmd->name = (char *) malloc(strlen(cmd->argv[0])+1);

    if (NULL == cmd->name) { /* Verify malloc */
        fprintf(stderr, "launch: Error allocating memory for command name %s: %s\n"
,
                cmd->argv[0], strerror(errno));
        return EXIT_FAILURE;
    }

    strcpy(cmd->name, cmd->argv[0]);

    return EXIT_SUCCESS;
}

/* Frees dynamically allocated strings from a command. */
void free_command(command_t *cmd) {
    int i;
    for (i=0; ((i < cmd->argc) && (cmd->argv[i] != NULL)); i++) {
        free(cmd->argv[i]);
    }
    free(cmd->name);
}

/* Given the name of a file containing a list of commands,
 * executes the commands in the file.
 * Returns EXIT_SUCCESS if all commands succeed, otherwise returns EXIT_FAILURE
 */
int launch_commands(const char *filename) {
    int i;
    int pid, num_children;
    int status;
    FILE* file;
    char cmdline[MAX_LINE_LEN];
    command_t command;

    /* Open the file that contains the set of commands. */
    file = fopen(filename, "r");
    if (!file) {
        fprintf(stderr, "launch: Error opening file %s: %s\n", filename,
                strerror(errno));
        return EXIT_FAILURE;
    }

    /* Report the process id. */
    printf("launch: Parent process id is %d\n", getpid());

    /* Process each command in the launch file. */
    num_children = 0;
    while (fgets(cmdline, MAX_LINE_LEN, file)) {

      if (parse_command(cmdline, &command) == EXIT_FAILURE)
        return EXIT_FAILURE;

        /* Create a child process to execute the command. */
        pid = fork();
        if (pid == 0) {
            /* The child executes the command. */
            execv(command.name, command.argv);
            /* If execv returns, there was an error */
            fprintf(stderr, "launch: Error executing command ’%s’: %s\n",
                    command.name, strerror(errno));
            return EXIT_FAILURE;
        } else if (pid < 0) {
            fprintf(stderr, "launch: Error while forking: %s\n", strerror(errno));
            return EXIT_FAILURE;
        }

        /* if pid > 0, then this is the parent process and there was
         * no error.  The parent reports the pid of the child process.
         */
        printf("launch: Forked child process %d with command ’%s’\n",
               pid, command.name);
        num_children++;

        /* The parent frees dynamically allocated memory in the
         * command data structure and continues to the next command.
         */
        free_command(&command);
    }
    if (ferror(file)) {
        perror("launch: Error while reading from file");
        return EXIT_FAILURE;
    }

    /* The parent closes the file. */
    if (fclose(file))
      perror("launch: Error closing file");

    printf("launch: Launched %d commands\n", num_children);

    /* The parent terminates after all children have terminated. */
    for (i=0; i < num_children; i++) {
        pid = wait(&status);
        if (pid < 0) {
            fprintf(stderr,
                    "launch: Error while waiting for child to terminate\n");
            return EXIT_FAILURE;
        } else  {
            printf("launch: Child %d terminated\n", pid);
        }
    }
    printf("launch: Terminating successfully\n");
    return EXIT_SUCCESS;
}


int main(int argc, char* argv[]) {

    /* Entry point for the testrunner program */
    if (argc > 1 && !strcmp(argv[1], "-test")) {
        run_launch_tests(argc - 1, argv + 1);
        return EXIT_SUCCESS;
    }

    /* Read the command-line parameters for this program. */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <launch_set_filename>\n",argv[0]);
        return EXIT_FAILURE;
    }

    /* Engage! */
    return launch_commands(argv[1]);
}
```