```c
/*
    Laboration 2: "miniShell 2.01"
    2G1520 Operating Systems
    Wilhelm Svenselius <wsv@kth.se>

    As usual, I think I overdid this, but I managed to do it without having to use
    a signal handler (wasn't compulsory, was it?). Enjoy. ^o^

    Compiles w/o errors or warnings using gcc -xc -Wall.
*/

/*
    Includes.
*/
#include <sys/types.h>      /* needed for pid_t */
#include <sys/wait.h>       /* needed for wait() and status codes */
#include <sys/time.h>       /* needed for gettimeofday */
#include <signal.h>         /* needed for kill() */
#include <stdlib.h>         /* needed for calloc() */
#include <stdio.h>          /* needed for fputs(), fgets() */
#include <string.h>         /* needed for strlen() */
#include <unistd.h>         /* needed for chdir() */
#include <ctype.h>          /* needed for isspace() */

/*
    Defines and symbolic constants.
*/
#define MAX_CMDLINE_LENGTH      ( 70 )
#define BOOL                    int
#define TRUE                    ( 1 )
#define FALSE                   ( 0 )
#define ISSPACE( x )            isspace( (int)( x ) )

/*
    Cleans up user input.
    - Turns all whitespace into regular spaces
    - Removes leading and trailing whitespace
    - Checks for control characters (such as escape characters) and complains
*/
BOOL cleanup( char* buf )
{
    const int len = strlen( buf );
    int i, ws = 0;

    /* Convert tabs to spaces and look for escape sequences */
    for( i = 0; i < len; i++ )
    {
        if( ISSPACE( buf[i] ) ) buf[i] = ' ';

        if( buf[i] < 32 )
        {
            puts( "*** Input contains unrecognized control character(s). Try again." );
            return FALSE;
        }
    }

    /* Remove trailing whitespace (includes newline added by fgets) */
    for( i = len - 1; i >= 0; i-- )
    {
        if( !ISSPACE( buf[i] ) )
            break;
        else
            buf[i] = 0;
    }

    /* Count leading whitespace */
    for( ws = 0; ISSPACE( buf[ws] ); ws++ );

    /* Remove leading whitespace */
    if( ws > 0 )
    {
        for( i = ws; buf[i] != 0; i++ )
            buf[i - ws] = buf[i];
        buf[i - ws] = 0;
    }

    return strlen( buf );
}

/*
    Given a command line, splits it up into an argument vector.
    Allocates memory for the argument vector and copies the arguments to it.
    Returns a pointer to the vector and sets the argument "argc" to the element count.
*/
char** argsplit( const char* buf, int* argc )
{
    char **argbuf;
    int a = 0, i, j, t, args = 1;
    BOOL inspace = TRUE;
    const int len = strlen( buf );

    /* Count arguments */
    for( i = 0; i < len; i++ )
    {
        if( !inspace && ISSPACE( buf[i] ) )
        {
            inspace = TRUE;
            args++;
        }
        else if( inspace && !ISSPACE( buf[i] ) )
            inspace = FALSE;
    }

    /* Allocate memory for buffer */
    argbuf = (char**) calloc( 1 + args, sizeof( char* ) );

    for( i = 0; i < len; i++ )
    {
        /* Skip whitespace and determine how much memory is needed */
        for( ; buf[i] != 0 && ISSPACE( buf[i] ); i++ );
        for( t = 0; buf[i+t] != 0 && !ISSPACE( buf[i+t] ); t++ );

        /* Allocate memory for argument 'a' and copy chars */
        argbuf[a] = (char*) calloc( 1 + t, sizeof( char ) );

        for( j = 0; j < t; j++ )
            argbuf[a][j] = buf[i+j];

        /* Advance counters */
        i += t; a++;
    }

    *argc = args;
    return argbuf;
}
/*
    Given a command line with no trailing or leading spaces,
    returns a pointer to the start of the argument list.
*/
const char* argstart( const char* buf )
{
    /* Find end of first argument */
    for( ; *buf != 0 && !ISSPACE( *buf ); buf++ );

    if( *buf != 0 )
    {
        /* Advance ptr to start of second argument */
        for( ; *buf != 0 && ISSPACE( *buf ); buf++ );
        return buf;
    }
    else
        return NULL;
}

/*
    Helper function, waits for a specific child process to end and prints the exit status.
*/
void wait_ch( pid_t cpid )
```

```c
{
    int st, cstat;

    /* Wait for child to exit */
    if( -1 != ( cpid = waitpid( cpid, &st, 0 ) ) )
    {
        if( WIFEXITED( st ) )
        {
            cstat = WEXITSTATUS( st );
            if( 0 != cstat )            /* Child exited by itself */
                printf( "*** Process [%ld] terminated with code: %d.\n", (long) cpid,     ↙
    cstat );
            else
                printf( "*** Process [%ld] terminated normally.\n", (long) cpid );
        }
        else if( WIFSIGNALED( st ) )    /* Child was terminated by signal */
            printf( "*** process [%ld] was terminated by signal: %d.\n", (long) cpid,     ↙
    WTERMSIG( st ) );
    }
    else
        perror( "waitpid" );
}

/*
    Main program.
*/
int main( void )
{
    BOOL user_exit = FALSE;
    char *cmdbuf, **argvec, *homedir;
    struct timeval statime, stotime;
    pid_t cpid1, cpid2;
    int arc;

    while( FALSE == user_exit )
    {
        /* Allocate memory for command buffer */
        cmdbuf = (char*) calloc( MAX_CMDLINE_LENGTH, sizeof( char ) );

        /* Show a pretty command prompt and read a line */
        fputs( "> ", stdout );
        fgets( cmdbuf, MAX_CMDLINE_LENGTH, stdin );

        /* Clean up and process */
        if( cleanup( cmdbuf ) )
        {
            /* Split into argument vector */
            argvec = argsplit( cmdbuf, &arc );

            if( strcmp( argvec[0], "exit" ) == 0 )
            {
                /* "exit */
                puts( "*** Goodbye." );
                user_exit = TRUE;
            }
            else if( strcmp( argvec[0], "cd" ) == 0 )
            {
                /* "cd" */
                if( -1 == chdir( argstart( cmdbuf ) ) )
                {
                    /* Try to change to home directory instead */
                    if( NULL != ( homedir = getenv( "HOME" ) ) )
                    {
                        puts( "*** Directory doesn't exist, trying home directory." );

                        if( -1 == chdir( homedir ) )
                            puts( "*** User's home directory is invalid!" );
                    }
                    else
                        perror( "chdir" );
                }
            }
            else if( strcmp( argvec[arc-1], "&" ) == 0 )
            {
                /* Background process (last argument was &) */
                if( 0 == ( cpid1 = fork() ) )
                {
                    /* Remove "&" from argument vector, increment "zombie counter" */
                    argvec[arc-1] = 0;

                    if( 0 == ( cpid2 = vfork() ) )
                    {
                        /* Try to execute program */
                        execvp( argvec[0], argvec );

                        /* We'll only get here if something went wrong */
                        perror( "execvp" );
                        exit( 1 );
                    }

                    /* Sentry code */
                    printf( "*** Background process [%ld] started.\n", (long) cpid2 );
                    wait_ch( cpid2 );
                    exit( 0 );
                }
            }
            else
            {
                /* Foreground process since nothing else matches */
                gettimeofday( &statime, NULL );

                if( 0 == ( cpid1 = vfork() ) )
                {
                    /* Try to execute program */
                    execvp( argvec[0], argvec );

                    /* We'll only get here if something went wrong */
                    perror( "execvp" );
                    exit( 1 );
                }
                else
                {
                    /* Parent code */
                    printf( "*** Foreground process [%ld] started.\n", (long) cpid1 );
                    wait_ch( cpid1 );

                    /* Print runtime */
                    gettimeofday( &stotime, NULL );
                    printf( "*** Runtime: %ld msec.\n", 1000 * ( stotime.tv_sec - statime.↙
    tv_sec ) + ( stotime.tv_usec - statime.tv_usec ) / 1000 );
                }
            }

            free( argvec ); argvec = NULL;
        }

        free( cmdbuf ); cmdbuf = NULL;
    }

    /* Kill off any remaining background processes as best we can! */
    kill( 0, SIGKILL );
    return 0;
}
```