ICT374 Assignment 2 Documentation

Table of Contents

[Project Declaration 3](#_Toc85128349)

[Extension 5](#_Toc85128350)

[List of Files 6](#_Toc85128351)

[Project Question 7](#_Toc85128352)

[Self Evaluation 10](#_Toc85128353)

[Fully Functional 10](#_Toc85128354)

[Not Functional 10](#_Toc85128355)

[Solution 11](#_Toc85128356)

[Test Evidence 12](#_Toc85128357)

[<case title> 12](#_Toc85128358)

[Purpose 12](#_Toc85128359)

[Output 12](#_Toc85128360)

[Results 12](#_Toc85128361)

[Source Code 13](#_Toc85128362)

[char\_vector.h 13](#_Toc85128363)

[char\_vector.c 14](#_Toc85128364)

[char\_vector\_vector.h 17](#_Toc85128365)

[char\_vector\_vector.c 18](#_Toc85128366)

[command.h 20](#_Toc85128367)

[command.c 21](#_Toc85128368)

[fileIORedirect.h 23](#_Toc85128369)

[fileIORedirect.c 24](#_Toc85128370)

[get\_line.h 25](#_Toc85128371)

[get\_line.c 25](#_Toc85128372)

[job.h 25](#_Toc85128373)

[job.c 27](#_Toc85128374)

[job\_vector.h 29](#_Toc85128375)

[job\_vector.c 31](#_Toc85128376)

[main.c 33](#_Toc85128377)

[PWDFuncs.h 33](#_Toc85128378)

[PWDFuncs.c 34](#_Toc85128379)

[tokenise.h 35](#_Toc85128380)

[tokenise.c 36](#_Toc85128381)

# Project Declaration



**Discipline of Information Technology,**

**Media and Communications**

**College of Arts, Business, Law and Social Sciences**

**ICT374 ASSIGNMENT 2**

**PROJECT DECLARATION**

**Group Members (full name and student number):**

**Member 1:** Rhys Mader 33705134

**Member 2:** Orlando Molina Santos 33302151

**Tutor’s Name:** Dr Hong Xie

**Assignment Due Date:** 29 October 2021

**Date Submitted:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Project Number (please tick):**

✓Project1: A Simple Unix Shell

🞎Project2: A Simple File Transfer Protocol

🞎Project 3: A Simple HTTP Client and Server

🞎Other Project (please specify): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Your assignment should meet the following requirements. Please confirm this (by ticking boxes) before submitting your assignment.**

* All details above are completed.
* We have read and understood the Documentation Requirements of this assignment
* **This assignment submission is compliant to the Documentation Requirements.**
* The archive file (a zip file) contains the file “Assignment2.docx”
* We have included all relevant Linux source code, executables and test files in the tar archive. The file names are chosen according to the project specification.
* This archive file will be submitted to ICT374 Unit LMS.
* We have kept a copy of this assignment, including this archive file, in a safe place.
* We have completed Task Allocation and Completion Record below.
* **We have signed the Group Declaration in the next page.**

**The unit coordinator may choose to use your submission as sample solutions to be viewed by other students, but only with your permission. Please indicate whether you give permission for this to be done.**

* Yes, we are willing to have my submission without change be made public as a sample solution.
* Yes, we are willing to have my submission be made public as a sample solution, as long as my submission is edited to remove all mentions of my identity.
* No, we are not willing to have my submission made public.

**Group Declaration**

As a group assignment, each member of the group is expected to make an equal contribution to the assignment and receives the same mark for the assignment.

However, we recognise that on some occasions and due to various reasons, the actual contributions to the assignment from the members could be unequal despite the best efforts of each member. In this case, we can still accept your assignment provided that all members of the group reach an agreement on their percentages of contribution to the assignment, and the agreement accurately reflects the real contribution by each member. In such a case, a member’s mark is linked to his or her agreed contribution to the assignment and is calculated using the following formula:

A member’s mark = minimum ( group mark x the member’s percentage of contribution x 2, group mark + 10, 100 )

On some rare occasions, the two members of the group fail to reach an agreement on their contributions to the assignment. In such a case, in order for your assignment to be marked, each member of the group must complete a detailed *Task Breakdown List* and state his or her own claim of the percentage of contribution to the assignment. Your tutor will then award each member a mark based on his assessment of the quality of the assignment as whole as well as his assessment of that member’s contribution to the assignment based on the information provided.

Please complete and sign ***one*** of the three declarations below:

|  |  |  |
| --- | --- | --- |
| *We have made* ***equal*** *contributions to this assignment. We understand that each of us will receive the same mark for this assignment.* | | |
| Signature (member 1): |  | Date: 20 Oct 2021 |
| Signature (member 2): |  | Date: 28 Oct 2021 |
| *We have made* ***unequal*** *contributions to this assignment. The percentage of contribution by each of us is given below (note the sum of the contributions by the two members must be equal to 100%):*  Member’s name: Contribution (%):  Member’s name: Contribution (%):  *We understand that each of us will receive a mark for this assignment that is linked to our contributions to the assignment. The mark will be calculated using the following formula:*  A member’s mark = minimum ( group mark x the member’s percentage of contribution x 2, group mark + 10, 100 )  Signature (member 1): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Signature (member 2): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | |
| *We are unable to reach an agreement on the percentage of our contributions to this assignment. However, in order for our tutor to be able to properly assess the work completed by each of us, each of us has completed a detailed Task Breakdown List which is included in this submission. We will accept our tutor’s determination of our contributions to this assignment.*  Signature (member 1): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Signature (member 2): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | |

# Extension

No extension has been granted.

# List of Files

* char\_vector.h
* char\_vector.c
* char\_vector\_vector.h
* char\_vector\_vector.c
* command.h
* command.c
* executeJob.h
* executeJob.c
* execute\_command.h
* execute\_command.c
* fileIORedirect.h
* fileIORedirect.c
* get\_line.h
* get\_line.c
* job.h
* job.c
* job\_vector.h
* job\_vector.c
* main.c
* pipe.h
* pipe.c
* PWDFuncs.h
* PWDFuncs.c
* tokenise.h
* tokenise.c

# Project Question

Design and implement a simple UNIX shell program using the grammar specified in the later part of this section. Please allow for at least 100 commands in a command line and at least 1000 arguments in each command.

In addition to the above, the following are required:

1. **Reconfigurable shell prompt (default %)**

The shell must have a shell built-in command prompt for changing the current prompt. For example, type the following command

% prompt john$

should change the shell prompt to john$, i.e., the second token of the command.

1. **The shell built-in command** pwd

This command prints the current directory (also known as working directory) of the shell process.

1. **Directory walk**

This command is similar to that provided by the Bash built-in command cd. In particular, typing the command without a path should set the current directory of the shell to the home directory of the user.

1. **Wildcard characters**

If a token contains wildcard characters \* or ?, the token is treated as a filename. The wildcard characters in such a token indicate to the shell that the filename must be expanded. For example the command

% ls \*.c

may be expanded to ls ex1.c ex2.c ex3.c if there are three matching files ex1.c ex2.c ex3.c in the current directory.

You may implement this feature using the C function glob.

1. **Standard input and output redirections > and <**

For example:

% ls -lt > foo

would redirect the standard output of the process ls -lt to file foo. Similarly, in the following command,

% cat < foo

the standard input of the process cat is redirected to file foo.

1. **Shell pipeline |**

For example:

% ls -lt | more

the standard output of the process ls -lt is connected to the standard input of the process more via a pipe.

1. **Background job execution**

For example:

% xterm &

The commannd line starts the command xterm in the background (i.e., the shell will not wait for the process to terminate and you can type in the next command immediately). The following command line

% sleep 20 & ps -l

starts the command sleep 20 and immediately execute the command ps -l without waiting for the command sleep 20 to finish first.

1. **Sequential job execution**

For example the command line

% sleep 20 ; ps -l

starts the command sleep 20 first, and wait for it to finish, then execute the command ps -l.

1. **The shell environment**

The shell should inherit its environment from its parent process.

1. **The shell built-in command exit**

Use the built-in command exit to terminate the shell program.

The behaviour of the above commands (except prompt) should be as close to those of the Bash shell as possible. In addition, your shell should not be terminated by CTRL-C, CTRL-\, or CTRL-Z.

Finally you must not use any existing shell program to implement your shell (for example by calling a shell through the function system). That would defeat the purpose of this project.

In the above, the commands such as ls, cat, grep, sleep, ps and xterm are used as examples to illustrate the use of your shell program. However, your shell must be able to handle any command or executable program. Note the commands prompt, pwd, cd and exit should be implemented as shell builtins, not as external commands.

The syntax and behaviour of the built-in commands pwd, cd and exit should be similar to the corresponding commands under Bash shell.

A major part of this shell is a command line parser. Please read the this note for suggestions on implementing the parser.

**Definition of Command Line Syntax**

The following is the formal definition of the command line syntax for the shell, defined in Extended BNF:

< command line > ::= < job >

| < job > '&'

| < job > '&' < command line >

| < job > ';'

| < job > ';' < command line >

< job > ::= < command >

| < job > '|' < command >

< command > ::= < simple command >

| < simple command > '<' < pathname >

| < simple command > '>' < pathname >

< simple command > ::= < pathname >

| < simple command > < token >

An informal definition plus additional explanations of the syntax is given below:

1. A ***command line*** consists of one or several ***jobs*** separated by the special character "&" and/or ";". The last ***job*** may be followed by the character "&" or ";". If a ***job*** is followed by the character "&", then it should be executed in the background.
2. A ***job*** consists of one or more ***commands*** separated by pipeline characters "|";
3. A ***command*** is either a ***simple command*** or a ***simple command*** followed by an input redirection (***< pathname***) or an output redirection (***> pathname***);
4. A ***simple command*** consists of a single ***pathname*** followed by zero or more tokens;
5. The following five characters are the ***special characters***: &, ;, |, < , > of the shell;
6. The ***white space characters*** are defined to be the space character and the tab character;
7. A ***token*** is a string. A string is a sequence of characters that does not normally contain space characters or special characters. If a string contains a space character or a special character, it must be preceded by the back slash character "\". In such a case, the back slash character and the space character, or the back slash character and the special character are intepreted as a single normal character without the special meaning. For example, the two character \& is interpreted as the normal character &. Note also that in this project we do not consider quoted strings. Therefore, if single quote or double quote characters appear in a string, they are treated just like any other non-special characters without its usually special meaning;
8. ***Tokens*** must be separated by one or more white spaces;
9. A ***pathname*** is either a file name, or an absolute pathname, or a relative pathname. Examples of pathnames are ***grep***, ***/usr/bin/grep***, ***bin/grep*** and ***./grep***;
10. A command line must end with a newline character.

# Self-Evaluation

## Fully Functional

* Tokenisation
* Command parsing
* Path wildcard expansion
* Character escaping
* Keyboard interrupt ignoring
* Exit command
* Prompt changing
* User line input
* Working directory changing
* Working directory printing
* Input redirection
* Output redirection
* Pipe creation
* Asynchronous job execution
* Sequential job execution

## Not Functional

* \_

# Solution

## Vectors

Whenever we needed to store a collection of homogenous data, we followed a vector pattern to create a structure to store this data and associated functions to manipulate these structures. Each vector stored its data in a dynamically allocated array alongside the current capacity, the maximum number of elements it can store before requiring a reallocation, and the current count, the number of currently stored elements. A reserve function is provided for each vector to allow for manual reallocations and thus reduce the number of reallocations that occur when multiple elements are appended to the vector using the append function, which was also provided. A safe element accessor, which performs an index range test before accessing an element, is also provided alongside functions to safely create and destroy each vector type.

These vectors are used extensively to structure the commands and jobs detected during the parsing of user input. They are also notably used to represent strings as character vectors which also has functions defined for it to convert between C-strings and our character vectors.

## User Input Parsing

Firstly, the user input is read into a character vector, keystroke by keystroke, until a newline character is detected, at which point this read line is passed to a tokeniser function. The tokeniser splits this string into whitespace separated tokens, stored as a character-vector vector, and then passes these tokens to the command parser. The command parser then constructs a vector of jobs which contain a vector of commands and an asynchronous flag with each command containing the command path, the command arguments which has the command path duplicated as the first argument, the input redirection file if given, and the output redirection file if given.

## Command Execution

We use a two-level process tree to manage the setup and execution of jobs and commands with the parent process responsible for:

* parsing commands,
* managing asynchronous job execution,
* creating pipes,
* and managing command execution

while the child processes are responsible for:

* expanding path wildcards,
* and prioritising special commands (exit, cd, prompt, pwd) before the execution of external commands.

By employing this structure, each running child process represents an outstanding command and these outstanding commands can easily signal their completion through their exit and consequent child exit signal to the parent process.

## Asynchronous Job Management

For jobs that are marked as asynchronous, forked child processes for each command in the job are added to a process group with the same ID as the PID of the first child process in the job. After all commands in a job have been executed, if the job is asynchronous, the execution continues without waiting for any child processes to finish. If the job is not asynchronous, execution is paused until there are no more processes in the shell’s process group.

## Pipe Management

The Pipe structure is used to encapsulate pipes. Functions for handling Pipes, including their creation, checking if read or write ends are open, and for closing read or write ends, are provided. Pipes are used to connect the standard output and standard input of child processes created for a job.

During the execution of a job, two pipe structures are tracked, the previous pipe and the next pipe, and both are initialised to NULL. When processing the commands in a job, the commands in the job are looped through, forking a child process for each one. Before forking, if there is a command to process after the current command, a Pipe is created on the heap, and stored as the next pipe. After forking:

* **In the parent process:** The parent process closes the write end of the next pipe, and the read end of the previous pipe. The previous pipe is deallocated, and then set to the next pipe. The next pipe is then set to NULL. Essentially, the next pipe for one child process becomes the previous pipe for the next child process.
* **In the child process:** The child process closes read end of the next pipe. If the write end of the next pipe is open, the standard output is redirected to it, and then the write end is closed, leaving only the standard output file descriptor referencing the underlying file. If the read end of the previous pipe is open, the standard input is redirected to it, and the read end is closed, leaving only the standard input file descriptor referencing the underlying file. Essentially, if the child process is the first process in a job with multiple commands, only its standard output is redirected to a pipe; if it is the last process, only its standard input is redirected; and if it falls between the first and the last, both its standard input and standard output are redirected.

## File IO Redirection

If a Command struct indicates that a process must have its standard input or standard output redirected to a file, the redirection occurs in the child process after any redirection to pipes has occurred, and before the command is executed. In the case of standard input redirection to an input file, if the specified file does not exist or cannot be opened in read-only mode, the process exits with an error. Otherwise, the standard input is redirected, and the file descriptor is closed, so that only the standard input file descriptor references the underlying file. In the case of standard output redirection to an output file, the process attempts to open the specified file in write-only mode. If the file does not exist, it is created with read/write permissions for the user and group, and read permissions for others. Once the file is open, the standard output is redirected to it, and the file descriptor is closed so that only the standard output file descriptor references the underlying file.

## Directory Walking

The directory walk only changes the shell working directory if the “cd” command is the only command in the job. If the command has more than one token, the first token is used as the argument pathname, otherwise the home directory as specified by the HOME environment variable is used. Then an attempt is made to change the current directory to the determined directory is made. If the working directory is successfully changed, an attempt is made to change the value of the PWD environment variable to match the current working directory. If the value of PWD cannot be updated to the new current working directory, the current working directory is changed back to the directory specified by PWD.

# Test Evidence

## Exit Command

### Purpose

Ensure the shell will exit when the exit command is entered.

### Output

\_

### Results

\_

## Print Working Directory

### Purpose

Ensure the shell will print the current directory when the print directory command is entered.

### Output

\_

### Results

\_

## Change Prompt

### Purpose

Ensure the shell will change the prompt when the change prompt command is entered and will save this change between sessions.

### Output

\_

### Results

\_

## Ignore Interrupt Signal

### Purpose

Ensure the shell will ignore keyboard interrupt signals.

### Output

\_

### Results

\_

## External Command

### Purpose

Ensure the shell will run an external command.

### Output

\_

### Results

\_

## Change Working Directory

### Purpose

Ensure the shell will change its working directory when the change working directory command is entered.

### Output

\_

### Results

\_

## Input Redirection

### Purpose

Ensure the shell will redirect the standard input of an external command to a given file.

### Output

\_

### Results

\_

## Output Redirection

### Purpose

Ensure the shell will redirect the standard output of an external command to a given file.

### Output

\_

### Results

\_

## External Command Arguments

### Purpose

Ensure the shell will forward command arguments to an external command, including the first argument as the command path.

### Output

\_

### Results

\_

## Pipes

### Purpose

Ensure the shell will connect commands in the same job with pipes correctly.

### Output

\_

### Results

\_

## Asynchronous Jobs

### Purpose

Ensure the shell will run jobs marked as background asynchronously, including the last job.

### Output

\_

### Results

\_

## Sequential Jobs

### Purpose

Ensure the shell will run jobs not marked as background sequentially, including the last job if it’s marked with a semi-colon or not marked.

### Output

\_

### Results

\_

# Source Code

## char\_vector.h

#ifndef CHAR\_VECTOR\_H

#define CHAR\_VECTOR\_H

struct CharVec {

char\* vec; ///The dynamic array containing all elements

unsigned int count; ///The number of elements on the vector

unsigned int capacity; ///The number of elements this vector can currently hold

};

/\*\* Create a new character vector

\return The new character vector (NULL if failed)

\*/

struct CharVec\* createCharVec();

/\*\* Create a new character vector from a C-string

\param str The C-string to copy character from (excluding terminating '\0')

\return The new character vector (NULL if failed)

\*/

struct CharVec\* createCharVecStr(const char\*const str);

/\*\* Destroy the given character vector

\param vec The character vector to destroy

\*/

void destroyCharVec(struct CharVec\* vec);

/\*\* Copy the source character vector into the destination character vector

\param dst The destination character vector to copy into

\param src The source character vector to copy from

\return 0 on failure, 1 on success

\*/

int copyCharVec(struct CharVec\*const dst, const struct CharVec\*const src);

/\*\* Get an element from the given character vector

\param vec The character vector to search

\param index The index at which to find the element

\return A pointer to the found element (NULL if out of range)

\*/

char\* getEleCharVec(const struct CharVec\*const vec, unsigned int index);

/\*\* Copy the contents of the given character vector into the given C-string

\param str A pointer to the dynamic C-string to fill

\param vec The character vector the copy from

\return 0 on failure, 1 on success

\*/

int getStrCharVec(char\*\*const str, const struct CharVec\*const vec);

/\*\* Reserve at least the given capacity in the given character vector

\param vec The character vector to reserve space for

\param cap The minimum number of elements the given vector must be able to store

\return 0 on failure, 1 on success

\*/

int reserveCharVec(struct CharVec\*const vec, unsigned int cap);

/\*\* Append an element to the given character vector

\param vec The character vector to append to

\param c The value to append

\return 0 on failure, and 1 on success

\*/

int appendEleCharVec(struct CharVec\*const vec, char c);

/\*\* Clear all elements in the given character vector

\param vec The character vector to clear

\return 0 on failure, 1 on success

\*/

int clearCharVec(struct CharVec\*const vec);

/\*\* Check if the given character vectors have the same content

\param u A character vector to compare

\param v The other character vector to compare

\return 1 if true, 0 if false

\*/

int equalCharVec(const struct CharVec\*const u, const struct CharVec\*const v);

#endif // CHAR\_VECTOR\_H

## char\_vector.c

#include "char\_vector.h"

#include <stdlib.h>

#include <string.h>

struct CharVec\* createCharVec()

{

struct CharVec\* vec = (struct CharVec\*)malloc(sizeof(struct CharVec));

if (vec == NULL)

return NULL;

vec->vec = NULL;

vec->capacity = 0;

vec->count = 0;

return vec;

}

struct CharVec\* createCharVecStr(const char\*const str)

{

struct CharVec\* vec = (struct CharVec\*)malloc(sizeof(struct CharVec));

if (vec == NULL)

return NULL;

unsigned int cap = strlen(str) + 1;

char\* cha = malloc(sizeof(char) \* cap);

if (cha == NULL)

return NULL;

vec->count = cap - 1;

vec->capacity = cap;

vec->vec = cha;

for (unsigned int i = 0; i < vec->count; ++i)

vec->vec[i] = str[i];

return vec;

}

void destroyCharVec(struct CharVec\* vec)

{

if (vec == NULL)

return;

free(vec->vec);

free(vec);

}

int copyCharVec(struct CharVec\*const dst, const struct CharVec\* src)

{

if (dst == NULL || src == NULL)

return 0;

if (!reserveCharVec(dst, src->count))

return 0;

for (unsigned int i = 0; i < src->count; ++i)

dst->vec[i] = src->vec[i];

dst->count = src->count;

return 1;

}

char\* getEleCharVec(const struct CharVec\*const vec, unsigned int index)

{

if (vec == NULL || index >= vec->count)

return NULL;

return vec->vec + index;

}

int getStrCharVec(char\*\*const str, const struct CharVec\*const vec)

{

if (str == NULL || vec == NULL)

return 0;

char\* cha = realloc(\*str, sizeof(char) \* (vec->count + 1));

if (cha == NULL)

return 0;

\*str = cha;

for (unsigned int i = 0; i < vec->count; ++i)

(\*str)[i] = vec->vec[i];

(\*str)[vec->count] = '\0';

return 1;

}

int reserveCharVec(struct CharVec\*const vec, unsigned int cap)

{

if (vec == NULL)

return 0;

if (cap <= vec->capacity)

return 1;

char\* cha = realloc(vec->vec, sizeof(char) \* cap);

if (cha == NULL)

return 0;

vec->vec = cha;

vec->capacity = cap;

return 1;

}

int appendEleCharVec(struct CharVec\*const vec, char c)

{

if (vec == NULL)

return 0;

if (vec->count + 1 > vec->capacity) if(!reserveCharVec(vec, vec->count + 1))

return 0;

vec->vec[vec->count] = c;

++(vec->count);

return 1;

}

int clearCharVec(struct CharVec\*const vec)

{

if (vec == NULL)

return 0;

if (vec->count == 0)

return 1;

free(vec->vec);

vec->vec = NULL;

vec->count = 0;

vec->capacity = 0;

return 1;

}

int equalCharVec(const struct CharVec\*const u, const struct CharVec\*const v)

{

if (u == NULL || v == NULL || u->count != v->count)

return 0;

for (unsigned int i = 0; i < u->count; ++i)

if (u->vec[i] != v->vec[i])

return 0;

return 1;

}

## char\_vector\_vector.h

#ifndef CHAR\_VECTOR\_VECTOR\_H

#define CHAR\_VECTOR\_VECTOR\_H

#include "char\_vector.h"

struct CharVecVec {

struct CharVec\* vec; ///The dynamic array of character vectors

unsigned int count; ///The number of character vectors in this vector

unsigned int capacity; ///The number of character vectors this vector can hold

};

/\*\* Create an empty vector of character vectors

\return A pointer to the created vector of character vectors (NULL on failure)

\*/

struct CharVecVec\* createCharVecVec();

/\*\* Destroy the given vector of character vectors

\param vec The vector to destroy

\*/

void destroyCharVecVec(struct CharVecVec\* vec);

/\*\* Reserve more space in the given vector for the given minimum capacity

\param vec The vector of character vectors to expand

\param cap The minimum required capacity

\return 0 on failure, 1 on success

\*/

int reserveCharVecVec(struct CharVecVec\*const vec, unsigned int cap);

/\*\* Copy the source vector into the destination vector

\param dst The vector of character vectors to copy into

\param src The vector of character vectors to copy from

\return 0 on failure, 1 on success

\*/

int copyCharVecVec(struct CharVecVec\*const dst, const struct CharVecVec\*const src);

/\*\* Test if the given vectors of character vectors are equal

\param u A vector to compare

\param v The other vector to compare

\return 1 if equal, 0 if not equal

\*/

int equalCharVecVec(const struct CharVecVec\*const u, const struct CharVecVec\*const v);

/\*\* Get an element from the given vector of character vectors

\param vec The vector to search

\param index The index of the element to retrieve

\return A pointer to the found element (NULL if not found)

\*/

struct CharVec\* getEleCharVecVec(struct CharVecVec\*const vec, unsigned int index);

/\*\* Append the given element to the end of the given vector

\param vec The vector to append to the end of

\param ele The character vector to append

\return 0 on failure, 1 on success

\*/

int appendEleCharVecVec(struct CharVecVec\*const vec, const struct CharVec\*const ele);

/\*\* Clear the given vector of all elements

\param vec The vector to clear

\return 0 on failure, 1 on success

\*/

int clearCharVecVec(struct CharVecVec\*const vec);

#endif // CHAR\_VECTOR\_VECTOR\_H

## char\_vector\_vector.c

#include "char\_vector\_vector.h"

#include <stdlib.h>

struct CharVecVec\* createCharVecVec()

{

struct CharVecVec\* vec = malloc(sizeof(struct CharVecVec));

if (vec == NULL)

return NULL;

vec->vec = NULL;

vec->count = 0;

vec->capacity = 0;

return vec;

}

void destroyCharVecVec(struct CharVecVec\* vec)

{

if (vec == NULL)

return;

for (unsigned int i = 0; i < vec->count; ++i)

destroyCharVec(vec->vec + i);

free(vec->vec);

free(vec);

vec = NULL;

}

int reserveCharVecVec(struct CharVecVec\*const vec, unsigned int cap)

{

if (vec == NULL)

return 0;

if (cap <= vec->capacity)

return 1;

struct CharVec\* cha = realloc(vec->vec, sizeof(struct CharVec) \* cap);

if (cha == NULL)

return 0;

vec->vec = cha;

for (unsigned int i = vec->count; i < cap; ++i)

{

(vec->vec + i)->vec = NULL;

(vec->vec + i)->capacity = 0;

(vec->vec + i)->count = 0;

}

vec->capacity = cap;

return 1;

}

int copyCharVecVec(struct CharVecVec\*const dst, const struct CharVecVec\*const src)

{

if (dst == NULL || src == NULL)

return 0;

if (!reserveCharVecVec(dst, src->count))

return 0;

for (unsigned int i = 0; i < src->count; ++i)

if (!copyCharVec(dst->vec + i, src->vec + i))

return 0;

dst->count = src->count;

return 1;

}

int equalCharVecVec(const struct CharVecVec\*const u, const struct CharVecVec\*const v)

{

if (u == NULL || v == NULL || u->count != v->count)

return 0;

for (unsigned int i = 0; i < v->count; ++i)

if (!equalCharVec(u->vec + i, v->vec + i))

return 0;

return 1;

}

struct CharVec\* getEleCharVecVec(struct CharVecVec\*const vec, unsigned int index)

{

if (vec == NULL || index >= vec->count)

return NULL;

return vec->vec + index;

}

int appendEleCharVecVec(struct CharVecVec\*const vec, const struct CharVec\*const ele)

{

if (vec == NULL || ele == NULL)

return 0;

if (!reserveCharVecVec(vec, vec->count + 1))

return 0;

if (!copyCharVec(vec->vec + vec->count, ele))

return 0;

(vec->count)++;

return 1;

}

int clearCharVecVec(struct CharVecVec\*const vec)

{

if (vec == NULL)

return 0;

for (unsigned int i = 0; i < vec->count; ++i)

destroyCharVec(vec->vec + i);

free(vec->vec);

vec->vec = NULL;

vec->capacity = 0;

vec->count = 0;

return 1;

}

## command.h

#ifndef COMMAND\_H\_INCLUDED

#define COMMAND\_H\_INCLUDED

#include "char\_vector\_vector.h"

#include "char\_vector.h"

struct Command

{

struct CharVec\* path; ///The path to search for the command

struct CharVecVec\* args; ///The vector of arguments to use

struct CharVec\* input\_file; ///The path to the input file to use (NULL for terminal or pipe)

struct CharVec\* output\_file; ///The path to the output file to use (NULL for terminal or pipe)

struct Command\* input\_pipe; ///The command to take input from (NULL for terminal or file)

struct Command\* output\_pipe; ///The command to output to (NULL for terminal or file)

};

/\*\* Write empty values to the given command

\param com The command to write empty values in

\return The given pointer on success or NULL on failure

\warning Does not properly destroy existing values

\note Should only be used when reserving space for a command

\*/

struct Command\* emplaceCom(struct Command\* com);

/\*\* Create an empty command

\return The new empty command

\*/

struct Command\* createCom();

/\*\* Destroy the given command

\param com The command to destroy

\*/

void destroyCom(struct Command\* com);

/\*\* Copy a given command into another given command

\param dst The command to copy into

\param src The command to copy from

\return 0 on failure, 1 on success

\note The input and output pipe members are only copied shallowly

\*/

int copyCom(struct Command\* dst, const struct Command\* src);

/\*\* Check if the given commands are equal

\param com1 A command to compare

\param com2 The other command to compare

\return 1 if equal, 0 if not equal

\*/

int equalCom(const struct Command\* com1, const struct Command\* com2);

#endif // COMMAND\_H\_INCLUDED

## command.c

#include "command.h"

#include <stdlib.h>

struct Command\* emplaceCom(struct Command\* com)

{

if (com == NULL)

return NULL;

com->path = NULL;

com->args = NULL;

com->input\_file = NULL;

com->output\_file = NULL;

com->input\_pipe = NULL;

com->output\_pipe = NULL;

return com;

}

struct Command\* createCom()

{

struct Command\* com = malloc(sizeof(struct Command));

if (com == NULL)

return NULL;

emplaceCom(com);

com->path = createCharVec();

com->args = createCharVecVec();

return com;

}

void destroyCom(struct Command\* com)

{

if (com == NULL)

return;

destroyCharVec(com->path);

destroyCharVecVec(com->args);

if (com->input\_file != NULL)

destroyCharVec(com->input\_file);

if (com->output\_file != NULL)

destroyCharVec(com->output\_file);

if (com->input\_pipe != NULL)

com->input\_pipe->output\_pipe = NULL;

if (com->output\_pipe != NULL)

com->output\_pipe->input\_pipe = NULL;

free(com);

}

int copyCom(struct Command\* dst, const struct Command\* src)

{

if (dst == NULL || src == NULL)

return 0;

if (dst->path == NULL)

dst->path = createCharVec();

if (dst->args == NULL)

dst->args = createCharVecVec();

if (!copyCharVec(dst->path, src->path))

return 0;

if (!copyCharVecVec(dst->args, src->args))

return 0;

if (src->input\_file == NULL)

{

if (dst->input\_file != NULL)

destroyCharVec(dst->input\_file);

dst->input\_file = NULL;

}

else

{

if (dst->input\_file == NULL)

dst->input\_file = createCharVec();

if (!copyCharVec(dst->input\_file, src->input\_file))

return 0;

}

if (src->output\_file == NULL)

{

if (dst->output\_file != NULL)

destroyCharVec(dst->output\_file);

dst->output\_file = NULL;

}

else

{

if (dst->output\_file == NULL)

dst->output\_file = createCharVec();

if (!copyCharVec(dst->output\_file, src->output\_file))

return 0;

}

dst->input\_pipe = src->input\_pipe;

dst->output\_pipe = src->output\_pipe;

return 1;

}

int equalCom(const struct Command\* com1, const struct Command\* com2)

{

return com1 != NULL

&& com2 != NULL

&& com1->input\_pipe == com2->input\_pipe

&& com1->output\_pipe == com2->output\_pipe

&& equalCharVec(com1->path, com2->path)

&& equalCharVecVec(com1->args, com2->args)

&& ((com1->input\_file == NULL && com2->input\_file == NULL)

|| (com1->input\_file != NULL && equalCharVec(com1->input\_file, com2->input\_file)))

&& ((com1->output\_file == NULL && com2->output\_file == NULL)

|| (com1->output\_file != NULL && equalCharVec(com1->output\_file, com2->output\_file)));

}

## execute\_command.h

#pragma once

#include "command.h"

/\*\* The keyword used to run the print working directory command \*/

#define PWD\_KEYWORD "pwd"

/\*\* The keyword used to run the change directory command \*/

#define CD\_KEYWORD "cd"

/\*\* The keyword used to run the change prompt command \*/

#define PROMPT\_KEYWORD "prompt"

/\*\* Execute the given command as the current process

\param com The command to run

\warning Does not return

\note Process will exit with -1 on failure

\*/

void executeCommand(const struct Command\*const com);

/\*\* Executes special built-in commands that affect the shell.

\param com: a Command struct to execute.

\return int: 0 if special commands not executed, or 1 if executed successfully.

\*/

int executeSpecial(const struct Command \*const com);

## execute\_command.c

#include "execute\_command.h"

#include "PWDFuncs.h"

#include "change\_prompt.h"

#include <unistd.h>

#include <stdlib.h>

int equalCharVecStr(struct CharVec \*vec, const char \*str)

{

int equal = 0;

struct CharVec \*chVec = createCharVecStr(str);

equal = equalCharVec(vec, chVec);

destroyCharVec(chVec);

return equal;

}

int executeSpecial(const struct Command \*const com)

{

if(com == NULL)

{

return 0;

}

char\* str = NULL;

int res;

//Note: a check for CD\_KEYWORD is in the normal executeCommand function

//but it wouldn't really do anything if called from a child process. That

//is OK and in line with the behaviour of the bash shell.

if(equalCharVecStr(com->path, CD\_KEYWORD))

{

if(com->args->count >= 2 && !getStrCharVec(&str, getEleCharVecVec(com->args, 1)))

{

return 0;

}

res = DirectoryWalk(str);

free(str);

if (res)

{

return 1;

}

}

return 0;

}

void executeCommand(const struct Command\*const com)

{

if (com == NULL)

exit(-1);

//check for special commands

char\* str = NULL;

int res;

if (equalCharVecStr(com->path, PWD\_KEYWORD))

{

PrintPWD();

exit(0);

}

if (equalCharVecStr(com->path, CD\_KEYWORD))

{

res = com->args->count < 2 || !getStrCharVec(&str, com->args->vec + 1) || !DirectoryWalk(str);

free(str);

if (res)

exit(0);

exit(-1);

}

if (equalCharVecStr(com->path, PROMPT\_KEYWORD))

{

res = com->args->count < 2 || !getStrCharVec(&str, com->args->vec + 1) || !changePrompt(str);

free(str);

if (res)

exit(0);

exit(-1);

}

if (!getStrCharVec(&str, com->path))

exit(-1);

char\*\* argv = malloc(sizeof(char\*) \* (com->args->count + 1));

if (argv == NULL)

{

free(str);

exit(-1);

}

argv[com->args->count] = NULL;

char\* arg = NULL;

for (unsigned int i = 0; i < com->args->count; ++i)

{

argv[i] = NULL;

if (!getStrCharVec(&argv[i], &(com->args->vec[i])))

{

free(str);

exit(-1);

}

}

execv(str, argv);

free(str);

exit(-1);

}

## executeJob.h

/\* File: executeJob.h

\* Date: 16 October 2021

\* Author: Orlando Molina Santos, ID: 33302151

\* Purpose: Provides functions to execute jobs from a command line.

\*/

#include "job.h"

/\* Executes a job from the command line.

\* job: Job struct representing the job to be executed.

\* Returns int: 0 if error, 1 on success

\*/

int ExecuteJob(const struct Job \*job);

## executeJob.c

/\* File: executeJob.c

\* Date: 16 October 2021

\* Author: Orlando Molina Santos, ID: 33302151

\* Purpose: Provides functions to execute jobs from a command line.

\*/

#include <stdlib.h>

#include <unistd.h>

#include "executeJob.h"

#include "pipe.h"

#include "fileIORedirect.h"

#include "execute\_command.h"

/\* Redirects stdin and stdout

\* prevPipe: Pipe to receive input from.

\* nextPipe: Pipe to redirect output to.

\* Pre: Is invoked in a child process.

\* Post: stdin and stdout are redirected to pipes.

\*/

void RedirectPipes(Pipe \*prevPipe, Pipe \*nextPipe)

{

CloseReadEnd(nextPipe);

CloseWriteEnd(prevPipe);

if(WriteEndOpen(nextPipe) && dup2(GetWriteEnd(nextPipe), STDOUT\_FILENO) == -1)

{

exit(1);

}

CloseWriteEnd(nextPipe);

if(ReadEndOpen(prevPipe) && dup2(GetReadEnd(prevPipe), STDIN\_FILENO) == -1)

{

exit(1);

}

CloseReadEnd(prevPipe);

}

/\* Redirects stdin and stdout

\* prevPipe: Pipe to receive input from.

\* nextPipe: Pipe to redirect output to.

\* command: the Command struct to redirect stdin and stdout for.

\* Pre: Is invoked in a child process

\* Post: stdin and stdout are redirected for child process

\*/

void RedirectStdinStdout(Pipe \*prevPipe, Pipe \*nextPipe, const struct Command \*command)

{

char \*fileOutBuf = NULL;

char \*fileInBuf = NULL;

if(command->input\_file != NULL)

{

if(getStrCharVec(&fileInBuf, command->input\_file) == 0)

{

exit(1);

}

if(access(fileInBuf, F\_OK) == -1)

{

printf("Shell: %s is not an existing file\n", fileInBuf);

exit(1);

}

}

if(command->output\_file != NULL && getStrCharVec(&fileOutBuf, command->output\_file) == 0)

{

exit(1);

}

RedirectPipes(prevPipe, nextPipe);

if(fileInBuf != NULL && RedirectStdin(fileInBuf) == 0)

{

exit(1);

}

if(fileOutBuf != NULL && RedirectStdout(fileOutBuf) == 0)

{

exit(1);

}

free(fileInBuf);

fileInBuf = NULL;

free(fileOutBuf);

fileOutBuf = NULL;

}

/\* Waits for all jobs in the same process group as caller to finish.

\*/

void WaitForSeqJob()

{

int status;

int returnPID = 0;

while(returnPID != -1)

{

returnPID = waitpid(0, &status, 0);

}

}

int ExecuteJob(const struct Job \*job)

{

Pipe \*nextPipe = NULL;

Pipe \*prevPipe = NULL;

int pid;

int pgID = -1;

//Some commands aren't executed in a forked child process.

if(job->count == 1 && executeSpecial(&(job->coms[0])))

{

return 1;

}

for(unsigned int i = 0; i < job->count; i++)

{

if(i + 1 < job->count && (nextPipe = CreatePipe()) == NULL)

{

return 0;

}

pid = fork();

if(pid == -1)

{

return 0;

}

if(pid == 0)

{

RedirectStdinStdout(prevPipe, nextPipe, &(job->coms[i]));

executeCommand(&(job->coms[i]));

}

CloseWriteEnd(prevPipe);

CloseReadEnd(prevPipe);

free(prevPipe);

prevPipe = NULL;

CloseWriteEnd(nextPipe);

prevPipe = nextPipe;

nextPipe = NULL;

if(job->async == 1)

{

//Set the process group ID for the job

if(pgID == -1)

{

pgID = pid;

if(setpgid(pid, 0) == -1)

{

return 0;

}

}

//Set the process group for the child process

if(setpgid(pid, pgID) == -1)

{

return 0;

}

}

}

if(job->async == 0)

{

WaitForSeqJob();

}

return 1;

}

## fileIORedirect.h

/\* File: fileIORedirect.h

\* Author: Orlando Molina Santos, ID: 33302151

\* Date: 28 September 2021

\* Purpose: Provides functions to redirect input and output.

\*/

/\* Redirects the input of a process from stdin.

\* inputFileName: the name of the input file to redirect to.

\* Returns: 0 if error, 1 on success

\*/

int RedirectStdin(const char \*inputFileName);

/\* Redirects the output of a process from stdout.

\* outputFileName: the name of the output file to redirect to.

\* Returns: 0 if error, 1 on success.

\*/

int RedirectStdout(const char \*outputFileName);

## fileIORedirect.c

/\* File: fileIORedirect.c

\* Author: Orlando Molina Santos, ID: 33302151

\* Date: 29 September 2021

\* Changes: 12 October 2021:

\* - Opened files are closed after stdin/stdout are redirected.

\* - Added appropriate permissions for created output files.

\* Purpose: Provides functions to redirect input and output.

\*/

#include <unistd.h>

#include <fcntl.h>

#include "fileIORedirect.h"

int RedirectStdin(const char \*inputFileName)

{

int success = 0;

int fd;

if((fd = open(inputFileName, O\_RDONLY)) == -1)

{

return 0;

}

if(dup2(fd, STDIN\_FILENO) >= 0)

{

success = 1;

}

close(fd);

return success;

}

int RedirectStdout(const char \*outputFileName)

{

int success = 0;

int fd;

//mode of 0664 is rw-rw-r--, which is what bash shell sets for new output files.

if((fd = open (outputFileName, O\_WRONLY|O\_TRUNC|O\_CREAT, 0664)) == -1)

{

return 0;

}

if(dup2(fd, STDOUT\_FILENO) >= 0)

{

success = 1;

}

close(fd);

return success;

}

## get\_line.h

#ifndef GET\_LINE\_H\_INCLUDED

#define GET\_LINE\_H\_INCLUDED

#include "char\_vector.h"

/\*\* Get a line from standard input

\param v The character vector to fill with the retrieved line

\return 0 on failure, 1 on success

\note Blocks until a newline or end of file character has been read

\*/

int getLine(struct CharVec\* v);

#endif // GET\_LINE\_H\_INCLUDED

## get\_line.c

#include "get\_line.h"

#include "char\_vector.h"

#include <stdio.h>

int getLine(struct CharVec\* v)

{

if (!clearCharVec(v)) return 0;

int c;

while (1)

{

c = getchar();

if (c == EOF || c == '\n') break;

if (!appendEleCharVec(v, (char)c)) return 0;

}

return 1;

}

## job.h

#ifndef JOB\_H\_INCLUDED

#define JOB\_H\_INCLUDED

#include "command.h"

struct Job

{

struct Command\* coms; ///The dynamic array that hollds all the command elements

unsigned int capacity; ///The number of elements the job can hold

unsigned int count; ///The number of elements the job is currently holding

unsigned char async; ///0 if this job should be executed sequentially, 1 if this job should be executed asynchronously

};

/\*\* Write empty values to the given job

\param job The job to write empty values in

\return The given pointer on success or NULL on failure

\warning Does not properly destroy existing values

\note Should only be used when reserving space for a job

\*/

struct Job\* emplaceJob(struct Job\* job);

/\*\* Create an empty job

\return The empty job

\*/

struct Job\* createJob();

/\*\* Destroy the given job

\param job The job to destroy

\*/

void destroyJob(struct Job\* job);

/\*\* Copy the given job into the other given job

\param dst The job to copy into

\param src The job to copy from

\return 0 on failure, 1 on success

\note Keeps command pipe pointers relative to itself rather than exactly copying them

\note (thus source and destination will fail value equal comparison)

\*/

int copyJob(struct Job\*const dst, const struct Job\*const src);

/\*\* Test if the given jobs are equal

\param job1 A job to compare

\param job2 The other job to compare

\return 1 if equal, 0 if not equal

\*/

int equalJob(const struct Job\*const job1, const struct Job\*const job2);

/\*\* Expand the capacity of the given job to hold at least the given number of commands

\param job The job to expand

\param cap The minimum number of elements to hold

\return 0 on failure, 1 on success

\*/

int reserveJob(struct Job\*const job, unsigned int cap);

/\*\* Add the given command onto the end of the given job

\param job The job to add onto

\param com The element to add

\return 0 on failure, 1 on success

\*/

int appendEleJob(struct Job\*const job, const struct Command\* com);

/\*\* Get the command in the given job at the given index

\param job The job to search

\param index The index at which to search

\return The command at the given index in the given job (NULL if not found)

\*/

struct Command\* getEleJob(const struct Job\*const job, unsigned int index);

/\*\* Clear the given job

\param job The job to clear

\*/

void clearJob(struct Job\*const job);

#endif // JOB\_H\_INCLUDED

## job.c

#include "job.h"

#include "command.h"

#include <stdlib.h>

struct Job\* emplaceJob(struct Job\* job)

{

if (job == NULL)

return NULL;

job->coms = NULL;

job->capacity = 0;

job->count = 0;

job->async = 0;

return job;

}

struct Job\* createJob()

{

return emplaceJob(malloc(sizeof(struct Job)));

}

void destroyJob(struct Job\* job)

{

if (job == NULL)

return;

for (int i = job->count - 1; i > 0 ; --i)

destroyCom(job->coms + i);

free(job->coms);

free(job);

}

int copyJob(struct Job\*const dst, const struct Job\*const src)

{

if (dst == NULL || src == NULL)

return 0;

if (!reserveJob(dst, src->count))

return 0;

for (unsigned int i = 0; i < src->count; ++i)

if (!copyCom(dst->coms + i, src->coms + i))

return 0;

for (unsigned int i = 0; i < src->count; ++i)

{

if (src->coms[i].input\_pipe >= src->coms && src->coms[i].input\_pipe < src->coms + src->count)

dst->coms[i].input\_pipe = dst->coms + (src->coms[i].input\_pipe - src->coms);

if (src->coms[i].output\_pipe >= src->coms && src->coms[i].output\_pipe < src->coms + src->count)

dst->coms[i].output\_pipe = dst->coms + (src->coms[i].output\_pipe - src->coms);

}

dst->count = src->count;

dst->async = src->async;

return 1;

}

int equalJob(const struct Job\*const job1, const struct Job\*const job2)

{

if (job1 == NULL || job2 == NULL || job1->count != job2->count || job1->async != job2->async)

return 0;

for (unsigned int i = 0; i < job1->count; ++i)

if (!equalCom(job1->coms + i, job2->coms + i))

return 0;

return 1;

}

int reserveJob(struct Job\*const job, unsigned int cap)

{

if (job == NULL)

return 0;

if (cap <= job->capacity)

return 1;

struct Command\* temp = realloc(job->coms, sizeof(struct Command) \* cap);

if (temp == NULL)

return 0;

job->coms = temp;

for (unsigned int i = job->count; i < cap; ++i)

emplaceCom(job->coms + i);

job->capacity = cap;

return 1;

}

int appendEleJob(struct Job\*const job, const struct Command\* com)

{

if (job == NULL || com == NULL)

return 0;

if (!reserveJob(job, job->count + 1))

return 0;

if (!copyCom(job->coms + job->count, com))

return 0;

++(job->count);

return 1;

}

struct Command\* getEleJob(const struct Job\*const job, unsigned int index)

{

if (job == NULL || index >= job->count)

return NULL;

return job->coms + index;

}

void clearJob(struct Job\*const job)

{

if (job == NULL)

return;

for (unsigned int i = 0; i < job->count; ++i)

destroyCom(job->coms + i);

free(job->coms);

emplaceJob(job);

}

## job\_vector.h

#ifndef JOB\_VECTOR\_H\_INCLUDED

#define JOB\_VECTOR\_H\_INCLUDED

#include "job.h"

struct JobVec

{

struct Job\* vec; ///The dynamic vector containing all jobs

unsigned int capacity; ///The maximum number of jobs this can currently hold

unsigned int count; ///The number of jobs this is currently holding

};

/\*\* Construct an empty job vector

\return A pointer to the new job vector, or NULL on failure

\*/

struct JobVec\* createJobVec();

/\*\* Destruct the given job vector

\param vec A pointer to the the job vector to destruct

\*/

void destroyJobVec(struct JobVec\* vec);

/\*\* Ensure atleast the given number of jobs can be stored in the given job vector

\param vec The job vector to expand

\param cap The minimum capacity required

\return 0 on failure, 1 on success

\*/

int reserveJobVec(struct JobVec\*const vec, unsigned int cap);

/\*\* Copy the source job vector into the destination

\param dst The destination to copy into

\param src The source to copy from

\return 0 on failure, 1 on success

\*/

int copyJobVec(struct JobVec\*const dst, const struct JobVec\*const src);

/\*\* Compare the given job vectors for value equality

\param vec0 The first job vector to compare

\param vec1 The other job vector to compare

\return 1 if equal, 0 if not equal

\*/

int equalJobVec(const struct JobVec\*const vec0, const struct JobVec\*const vec1);

/\*\* Get an element from the given job vector at the given index

\param vec The job vector to search

\param index The index to search at

\return The job element in the given job vector at the given index, or NULL if index is out of range

\*/

struct Job\* getEleJobVec(const struct JobVec\*const vec, unsigned int index);

/\*\* Append the given element to the given job vector

\param vec The job vector to append to

\param ele The job to append

\return 0 on failure, 1 on success

\*/

int appendEleJobVec(struct JobVec\*const vec, const struct Job\*const ele);

/\*\* Clear the given job vector of elements and allocated memory

\param vec The job vector to clear

\return 0 on failure, 1 on success

\*/

int clearJobVec(struct JobVec\*const vec);

#endif // JOB\_VECTOR\_H\_INCLUDED

## job\_vector.c

#include "job\_vector.h"

#include <stdlib.h>

struct JobVec\* emplaceJobVec(struct JobVec\* vec)

{

if (vec == NULL)

return NULL;

vec->vec = NULL;

vec->capacity = 0;

vec->count = 0;

return vec;

}

struct JobVec\* createJobVec()

{

return emplaceJobVec(malloc(sizeof(struct JobVec)));

}

void destroyJobVec(struct JobVec\* vec)

{

if (vec == NULL)

return;

for (int i = vec->count; i > -1; --i)

destroyJob(vec->vec + i);

free(vec->vec);

free(vec);

}

int reserveJobVec(struct JobVec\*const vec, unsigned int cap)

{

if (vec == NULL)

return 0;

if (vec->capacity >= cap)

return 1;

struct Job\* v = realloc(vec->vec, sizeof(struct Job) \* cap);

if (v == NULL)

return 0;

vec->vec = v;

for (unsigned int i = vec->count; i < cap; ++i)

if (emplaceJob(vec->vec + i) == NULL)

return 0;

vec->capacity = cap;

return 1;

}

int copyJobVec(struct JobVec\*const dst, const struct JobVec\*const src)

{

if (dst == NULL || src == NULL)

return 0;

if (!reserveJobVec(dst, src->count))

return 0;

for (unsigned int i = 0; i < src->count; ++i)

if (!copyJob(dst->vec + i, src->vec + i))

return 0;

dst->count = src->count;

return 1;

}

int equalJobVec(const struct JobVec\*const vec0, const struct JobVec\*const vec1)

{

if (vec0 == NULL || vec1 == NULL || vec0->count != vec1->count)

return 0;

for (unsigned int i = 0; i < vec0->count; ++i)

if (!equalJob(vec0->vec + i, vec1->vec + i))

return 0;

return 1;

}

struct Job\* getEleJobVec(const struct JobVec\*const vec, unsigned int index)

{

if (vec == NULL || index >= vec->count)

return NULL;

return vec->vec + index;

}

int appendEleJobVec(struct JobVec\*const vec, const struct Job\*const ele)

{

if (vec == NULL || ele == NULL)

return 0;

if (!reserveJobVec(vec, vec->count + 1))

return 0;

if (!copyJob(vec->vec + vec->count, ele))

return 0;

++(vec->count);

return 1;

}

int clearJobVec(struct JobVec\*const vec)

{

if (vec == NULL)

return 0;

free(vec->vec);

vec->vec = NULL;

vec->capacity = 0;

vec->count = 0;

return 1;

}

## main.c

#include "char\_vector.h"

#include "get\_line.h"

#include <stdio.h>

int main(int argc, char\*\* argv, char\*\* env)

{

const char\* PROMPT = ">>> "; //default prompt

struct CharVec\* line = createCharVec(); //holds input line

struct CharVec\* EXIT\_COM = createCharVecStr("exit"); //the exit command

while (1)

{

printf(PROMPT);

getLine(line);

if (equalCharVec(line, EXIT\_COM)) break;

//continue with command parsing

}

destroyCharVec(&line);

destroyCharVec(&EXIT\_COM);

return 0;

}

## pipe.h

/\* File: pipe.h

\* Date: 16 October 2021

\* Author: Orlando Molina Santos, ID: 33302151

\* Purpose: Provides a struct and functions for dealing with pipes.

\*/

/\* Encapsulates a pipe.

\*/

typedef struct

{

int read;

int write;

} Pipe;

/\* Creates a pipe.

\* Returns: NULL on error, or a Pipe on the heap if successful.

\*/

Pipe \*CreatePipe();

/\* Gets the file descriptor for the read end of a Pipe

\* Returns int: the file descriptor for the read end of the Pipe.

\*/

int GetReadEnd(Pipe \*pipe);

/\* Gets the file descriptor for the write end of a Pipe

\* Returns int: the file descriptor for the write end of the Pipe.

\*/

int GetWriteEnd(Pipe \*pipe);

/\* Closes the read end of a Pipe.

\*/

void CloseReadEnd(Pipe \*pipe);

/\* Closes the write end of a Pipe

\*/

void CloseWriteEnd(Pipe \*pipe);

/\* Checks if the read end of a Pipe is open

\* Returns int: 1 if read end is open, 0 if read end is closed.

\*/

int ReadEndOpen(Pipe \*pipe);

/\* Checks if the write end of a Pipe is open

\* Returns int: 1 if write end is open, 0 if write end is closed.

\*/

int WriteEndOpen(Pipe \*pipe);

## pipe.c

/\* File: pipe.c

\* Date: 16 October 2021

\* Author: Orlando Molina Santos, ID: 33302151

\* Purpose: Provides a struct and functions for dealing with pipes.

\*/

#include <unistd.h>

#include "pipe.h"

Pipe \*CreatePipe()

{

int tempPipe[2];

Pipe \*p = malloc(sizeof(Pipe));

if(p == NULL)

{

return NULL;

}

if(pipe(tempPipe) == 0)

{

p->read = tempPipe[0];

p->write = tempPipe[1];

}

else

{

free(p);

p = NULL;

}

return p;

}

int GetReadEnd(Pipe \*pipe)

{

if(pipe == NULL)

{

return -1;

}

return pipe->read;

}

int GetWriteEnd(Pipe \*pipe)

{

if(pipe == NULL)

{

return -1;

}

return pipe->write;

}

void CloseReadEnd(Pipe \*pipe)

{

if(pipe != NULL && pipe->read != -1)

{

close(pipe->read);

pipe->read = -1;

}

}

void CloseWriteEnd(Pipe \*pipe)

{

if(pipe != NULL && pipe->write != -1)

{

close(pipe->write);

pipe->write = -1;

}

}

int ReadEndOpen(Pipe \*pipe)

{

if(pipe == NULL)

{

return 0;

}

return (pipe->read >= 0);

}

int WriteEndOpen(Pipe \*pipe)

{

if(pipe == NULL)

{

return 0;

}

return (pipe->write >= 0);

}

## PWDFuncs.h

/\* File: PWDFuncs.h

\* Author: Orlando Molina Santos

\* Date: 2 October 2021

\* Purpose: Provides functions to change the current directory and print its pathname.

\*/

/\* Prints the PWD of the shell process.

\*/

void PrintPWD();

/\* Gets the path of the current working directory.

\* Returns char\*: NULL if error, otherwise points to a string on the heap containing the current working directory.

\* Remember to free the returned pointer when done.

\*/

char \*GetCWD();

/\* Changes the working directory, and sets the PWD.

\* pathname: the new working directory. If NULL, function will try to change directory to user's HOME directory.

\* Returns int: 0 if directory and PWD can't be changed, 1 on success.

\*/

int DirectoryWalk(const char \*pathname);

## PWDFuncs.c

/\* File: PWDFuncs.c

\* Author: Orlando Molina Santos

\* Date: 2 October 2021

\* Purpose: Provides functions for changing the current working directory and printing its path.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include "PWDFuncs.h"

extern char \*\*environ;

void PrintPWD()

{

printf("%s\n", getenv("PWD"));

}

char \*GetCWD()

{

// Allocate memory for buffer to receive current directory path.

size\_t bufSize = sizeof(char) \* 256;

char \*cwd = malloc(bufSize);

if(cwd == NULL)

{

return NULL;

}

// Try to get current working directory path.

// If buffer is too small to receive path, increase size and try again.

while(getcwd(cwd, bufSize) == NULL)

{

char \*temp;

bufSize += sizeof(char) \* 50;

temp = realloc(cwd, bufSize);

if(temp == NULL)

{

free(cwd);

cwd = NULL;

return NULL;

}

else

{

cwd = temp;

}

}

return cwd;

}

/\* Sets the PWD environment variable to the current working directory.

\* Returns int: 0 on error, 1 on success.

\*/

int SetPWDtoCWD()

{

int success = 0;

char \*cwd = GetCWD();

if(cwd != NULL)

{

//+1 because setenv returns -1 for fail, 0 for success

success = setenv("PWD", cwd, 1) + 1;

free(cwd);

}

return success;

}

int DirectoryWalk(const char \*pathname)

{

int success = 0;

if(pathname == NULL)

{

pathname = getenv("HOME");

}

if(chdir(pathname) == 0 && (success = SetPWDtoCWD()) == 0)

{

chdir(getenv("PWD"));

}

return success;

}

## tokenise.h

#ifndef TOKENISE\_H\_INCLUDED

#define TOKENISE\_H\_INCLUDED

#include "char\_vector\_vector.h"

#include "char\_vector.h"

#define TOKEN\_SEP ' ' /// The character used to separate tokens

/\*\* Split the given character vector into a vector of space separated tokens

\param out The vector to fill with tokens

\param str The character vector to split into tokens

\return 0 on failure, 1 on success

\*/

int tokenise(struct CharVecVec\*const out, const struct CharVec\*const str);

#endif // TOKENISE\_H\_INCLUDED

## tokenise.c

#include "tokenise.h"

#include <stdlib.h>

int tokenise(struct CharVecVec\*const out, const struct CharVec\*const str)

{

if (out == NULL || str == NULL)

return 0;

if (!clearCharVecVec(out))

return 0;

struct CharVec\* token = createCharVec();

int is\_token = 0;

for (unsigned int c = 0; c < str->count; ++c)

{

switch (str->vec[c])

{

case TOKEN\_SEP:

if (is\_token)

{

is\_token = 0;

if (!appendEleCharVecVec(out, token))

return 0;

}

break;

default:

if (!is\_token)

{

is\_token = 1;

if (!clearCharVec(token))

return 0;

}

if (!appendEleCharVec(token, str->vec[c]))

return 0;

}

}

if (is\_token)

if (!appendEleCharVecVec(out, token))

return 0;

return 1;

}