Functions

We can use some function that we already know (like `printf`, `malloc`, `free`, etc) so I will not describe them here.

You will probably not use all of these functions but at least you have somewhere where you can easily find links to the manual pages. And for some, an example on how to use them.

readline()

```
char *readline (const char *prompt);
```

The `readline()` function reads a line from the terminal and returns it, using `prompt` as a prompt. If no prompt is given as parameter, no prompt will be shown in the terminal. The line returned is allocated with `malloc` and we have to free it ourselves.

readline()

```c
#include <stdio.h>
#include <readline/readline.h>
#include <readline/history.h>

int main(void)
{
    char *rl;
    rl = readline("Prompt > ");
    printf("%s\n", rl);
    return (0);
}
```

Compiling this programm and running will result in the following.

```
$> ./minishell
Prompt > Hi ! How are you ?
Hi ! How are you ?
$>
```

You can find more information about `readline()` here.

rl_clear_history()

```c
void rl_clear_history(void);
```

The `rl_clear_line()` function clears the history list by deleting all of the entries. The `rl_clear_line()` function frees data that the `readline` library saves in the histroy list.

rl_on_new_line()

```c
int rl_on_new_line(void);
```

The `rl_on_new_line()` function tells the update routine that we have moved onto a new empty line, usually used after outputting a line.

rl_replace_line()

I didn't find any information on that function.

rl_redisplay()

```
int rl_redisplay(void);
```

The `rl_redisplay()` change what's displayed on the screen to reflect the current contents of `rl_line_buffer`.

add_history()

```
void add_history(char *s);
```

The `add_history()` function saves the line passed as parameter in the history so it can be retrieved later in the terminal (like pressing the up arrow in bash).

getcwd()

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

The `getcwd()` returns a null-terminated string containing the absolute pathname that is the current working directory of the calling process. The pathname is returned as the function result and via the argument `buf`.

getcwd() example

```c
#include <unistd.h>
#include <stdio.h> // for printf

int main(void)
{
    char *pwd;

    pwd = getcwd(NULL, 0);
    printf("pwd: %s\n", pwd);
    return (0);
}
```

```
$> pwd: /Users/saeby/Documents/tmp
```

You can find more information about `getcwd()` here.

chdir()

```c
#include <unistd.h>
int chdir(const char *path);
```

`chdir()` changes the current working directory of the calling process to the directory specified in `path`.

chdir() example

```c
#include <unistd.h>
#include <stdio.h> // for printf

int main(void)
{
    char *pwd;

    pwd = getcwd(NULL, 0);
    printf("pwd before chdir: %s\n", pwd);
    chdir("/Users/saeby/Documents/42/minishell");
    pwd = getcwd(NULL, 0);
    printf("pwd after chdir: %s\n", pwd);
    return (0);
}
```

```
$> pwd before chdir: /Users/saeby/Documents/tmp
$> pwd after chdir: /Users/saeby/Documents/42/minishell
```

You can find more information about `chdir()` here.

stat() & lstat() & fstat()

```c
#include <sys/stat.h>
int stat(const char *restrict pathname, struct stat *restrict statbuf);
int lstat(const char *restrict pathname, struct stat *restrict statbuf);
int fstat(int fd, struct stat *statbuf);
```

These functions return information about a file in the structure pointed to by `statbuf`.

You can find more detailed information about these functions here.

opendir()

```c
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

The `opendir()` function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

You can find more information about the `opendir` function here.

readdir()

```
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

The `readdir()` function returns a pointer to a `dirent` structure representing the next directory entry in the directory stream pointed to by `dirp`. It returns `NULL` on reaching the end of the directory stream or if an error occured.

You can find more information about `readdir` here.

closedir()

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

The `closedir()` function closes the directory stream associated with `dirp`. A successful call to `closedir()` also closes the underlying file descriptor associated with `dirp`. The directory stream descriptor `dirp` is not available after this call.

You can find more information about `closedir` here.

strerror()

```
#include <string.h>
char *strerror(int errnum);
```

The `strerror()` function returns a pointer to a string that describes the error code passed in the argument errnum. This string must not be modified by the application, but may be modified by a subsequent call to `strerror()` or `strerror_l()`. No other library function, including `perror()`, will modify this string.

You can find more information about `strerror` here.

perror()

```
#include <stdio.h>
void perror(const char *s);
```

The `perror()` function produces a message on standard error describing the last error encountered during a call to a system or library function.

You can find more information about `perror` here.

isatty()

```
#include <unistd.h>
int isatty(int fd);
```

The `isatty` function tests wether `fd` is a terminal.

You can find more information about `isatty` here.

ttyname()

```
#include <unistd.h>
char **ttyname(int fd);
```

The `ttyname()` function returns a pointer to the null-terminated pathname of the terminal device that is open on the file descriptor `fd`, or `NULL` on error.

You can find more information about `ttyname()` here.

ttyslot()

```
#include <unistd.h>
int ttyslot(void);
```

This is a legacy function with some backstory, you can read all about it and how it works here.

ioctl()

```
#include <sys/ioctl.h>
int ioctl(int fd, unsigned long request, ...);
```

The `ioctl()` system call manipulates the underlying device parameters of a special files. You can find more detailed information here.

getenv()

```
#include <stdlib.h>
char *getenv(const char *name);
```

The `getenv()` function searches the environment list to find the environment variable name, and returns a pointer to the corresponding value string.

You can find more information about `getenv()` here.

tcsetattr()

```
#include <termios.h>
int tcsetattr(int fildes, int optional_actions, const struct *termios_p);
```

The `tcsetattr()` function shall set the parameters associated with the terminal referred to by the open file descriptor `fildes` from the `termios` structure referenced by `termios_p` as described here.

tcgetattr()

```
#include <termios.h>
int tcgetattr(int fildes, struct termios *termios_p);
```

The `tcgetattr()` function shall get the parameters associated with with the terminal reffered to by `fildes` and store them in the `termios` structure referenced by `termios_p`.

You can find more detailed information here.

tgetent()

```
#include <curses.h>
#include <term.h>
int tgetent(char *bp, const char *name);
int tgetflag(char *id);
int tgetnum(char *id);
char *tgetstr(char *id, char **area);
char *tgoto(const char *cap, int col, int row);
int tputs(const char *str, int affcnt, int (*putc)(int));
```

These routines are included as a conversion aid for programs that use the `termcap` library. You can find more information about all of them here.

Previous
Understand Minishell

Next
Building the thing