Today we will continue writing our own Unix Shell. The features we plan to implement today are:

- command line options: --help, -h, --version, -v, -c
- CTRL-C resistant
- Environment variables (set, get) and their use

1. Preliminaries for options

We are going to have five options, that will execute three different actions:

- --help and -h will show a help message.
- --version and -v will show the shell version.
- -c will have an argument. This argument will be a command to be executed.

First, we need variables (flags) to identify which options were used. We will use one for each action. Notice that c_value will point to the command we are passing. If the option is not used, it remains NULL.

```c
static int h_flag = FALSE;
static int v_flag = FALSE;
static char * c_value = NULL;
```

Now, let us make the functions that each option will call. Including the following in the source code (457shell.c) after declaring all global variables and macros.

```c
void option_help()
{
    printf("457Shell – Custom UNIX Shell\n");
    printf("Usage: 457shell [long options] [option]\n");
    printf("long options:\n\t­­help\n\t­­version\n");
    printf("options:\n\t-c command\n");
    exit(0);
}

void option_version()
{
    printf("457Shell – Custom UNIX Shell – Version 1.26\n");
    exit(0);
}

void option_command()
{
    memset(command, 0x00, MAX_COMMAND_LENGTH);
    strncpy(command, c_value, MAX_COMMAND_LENGTH);
    parse_command();
    execute_command();
    exit(0);
}
```
2. Reading command line options using a for-loop

We will create a function to read all the options and our populate the option flags. Initially, we use the for-loop approach.

```c
int read_options(int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; i++) {
        if (strcmp(argv[i], "-h") || strcmp(argv[i], "--help")) {
            h_flag = TRUE;
        } else if (strcmp(argv[i], "-v") || strcmp(argv[i], "--version")) {
            v_flag = TRUE;
        } else if (strcmp(argv[i], "-c") == 0) {
            if (argc > i) {
                c_value = argv[++i];
            } else {
                printf("Error: Command is missing\n");
                exit(1);
            }
        } else {
            printf("Invalid option: %s\n", argv[i]);
            exit(1);
        }
    }
}
```

Now, our main function should initially call read_options and, when necessary, execute the respective actions.

Include the following code in the main function:

```c
read_options(argc, argv);
if (h_flag)
    option_help();
if (v_flag)
    option_version();
if (c_flag)
    option_command();
```

Compile and test these options, e.g.:

```
[user@term dir]$ ./457shell -v
[user@term dir]$ ./457shell --version
[user@term dir]$ ./457shell -h
[user@term dir]$ ./457shell --help
[user@term dir]$ ./457shell -c cd
```
3. Improving command line (using getopt_long)

In order to automate the processing of command line options, GNU C Library provides the functions “getopt” and “getopt_long”. While “getopt” is defined by “unistd.h”, “getopt_long” is defined in “getopt.h”. So, we need to include that header as well.

```c
#include <getopt.h>
```

And, the function to read the options using “getopt_long” is as follows:

```c
int get_options(int argc, char *argv[]) {
    int c;
    static struct option long_options[] = {
        {"version", no_argument, 0, 'v'},
        {"help", no_argument, 0, 'h'},
        {"command", required_argument, 0, 'c'},
        {0, 0, 0, 0}
    };
    int opt;

    while (1) {
        opt = getopt_long (argc, argv, "vhc:", long_options, NULL);
        /* Detect the end of the options. */
        if (opt == -1)
            break;

        switch (opt) {
            case 'v':
                v_flag=TRUE;
                break;
            case 'h':
                h_flag=TRUE;
                break;
            case 'c':
                c_value(optarg);
                break;
            case '?': // function getopt_long showed error message.
                break;
            default: // this should not happen
                exit(-1);
        }
    }
}
```

Now, change the “main” function to call “get_options” instead of “read_options”.

Notice that using getopt (or getopt_long, in our case), might initially seem more code. However, its use actually simplify the processing of command line options, making the code much more readable.
4. Becominig CTRL-C resistant

Using basic signal handling, we will avoid the user to quit our shell by pressing CTRL-C. When a user press CTRL-C, the OS sends a signal SIGINT to the program, so it can be interrupted. We will trap this signal by creating a function to handle that signal.

Functions that work as a “signal handler” should be of type “void” (i.e., should not return anything) and receive an integer as parameter. This integer is the signal identifier. Include our signal handler in our program.

```c
void signal_handler(int sig)
{
    if (sig == SIGINT) {
        printf(“\n”);
        memset(command, 0x00, MAX_COMMAND_LENGTH);
        if (write(1, PROMPT,sizeof(PROMPT))!=sizeof(PROMPT) ) {
            printf(“Error writing prompt\n”);
            exit(1);
        }
    }
}
```

Notice that, in a normal shell (e.g., bash), when we press CTRL-C, the prompt will be displayed again and we will type the new command, ignoring what was previously typed. Our shell will do the same.

To invoke the signal_handler, we need to include the following line to the main function:

```c
signal(SIGINT, &signal_handler);
```

The parameter &signal_handler indicates the memory address of the function.

5. Environment variables

Some commands use environment variables (such as $HOME) during its execution. We will now read and define values for these variables.

First, as we did with the other built-in commands, we need to define functions to perform these actions.

```c
void execute_get()
{
    char * value;
    if (argument_count > 1) {
        value=getenv(argument_list[1]);
        if (value == NULL)
            printf("Variable %s does not exist\n", argument_list[1]);
        else
            printf("%s=%s\n",argument_list[1],value);
```
Now, you need to modify the function “execute_command” to perform these two commands ("get" and “set”). Compile and verify if they are working:

cpsc 457 > get HOME
cpsc 457 > get TEST_VAR
cpsc 457 > set TEST_VAR this_value
cpsc 457 > get TEST_VAR

6. Our code is getting too long. Create a header file (457shell.h) with all the global declarations (variables, constants/macros) and change our source (457shell.c) to include that file.

   We need to change also the Makefile. It needs to check the header file for changes as well.

   Compile and verify if it is working.

7. (Optional) Since we can now access environment variables. Do the following modifications in our shell:
1. Change “cd” to go HOME when no parameter was informed
2. Include a built-in command “pwd” that shows the current directory. (Hint: Code is already implemented.)
3. (Challenge) Modify “execute_external” to look for an executable in the variable “PATH” before and use the program's absolute name, if found, to call execve. (Hints: Useful functions might include “strtok”, “strcat”, “open”.)

   NOTE: The source code (including the optional section) will be available for next tutorial.

8. (Question in the slides. To be asked at the end of today's session.)