UNIX 101

or "How to feel like a true hack3r"

Subject - C environment setup

Tuesday 30th November, 2021

Contents

1 Foreword

1.1 Notions seen in the tutorials

Welcome to this subject! If you have finished the first tutorial and the one on git (and I hope you do, if you have not yet, please finish them now), you should know:

- What the syntax of a command is
- A few useful commands
- How to navigate in a filesystem
- How to create, edit and remove files
- What is a packet manager and how to use one
- **How to create executable scripts**
- **The basics of git**

This is just enough to begin working on setting up an environment to write C code and handing out code for review.

1.2 Objectives

The goal of this subject is to make you use the notions seen in the tutorial to demonstrate the use of UNIX environments in "real-life" situations. By now, you should have basic knowledge on C programming. Do not worry if you only know basic still: the goal of this subject is to evaluate your skills in UNIX, not in C. The quality of your code is not going to be relevant.

2 Setup

Before beginning, make sure you:

- finished tutorial-1 and thus know how to write scripts
- finished tutorial-git and thus know how to regularily make commits and push your work
- have a Github account

Before starting anything, create a git repository named `c-environment`.

3 Instructions

The goal of this subject is to make you use standard UNIX tools to compile C code into actual programs. You should have already learned a bit of C in the past but you may never have compiled code on your own computer. Compiling is the act of transforming code (i.e. text) into an executable file. A big share of programming languages are compiled; the process to turn code into programs is similar to what we are going to learn for most of them.

The instructions for this subject are slim, on purpose. You are expected to lookup how to resolves the levels by yourself; but do not worry, the instructions are slim but so is the subject. You should be able to finish all levels entirely without trouble.

4 Evaluation

4.1 Instructions

You are asked to constituate a team of 2 persons. You will be graded as a team on:

- An oral presentation of your work (no need to prepare slides, we will discuss around your code)
- General C/UNIX knowledge questions

You have to push your work to your git repository before Wednesday 15th December, 2021, 7:00 AM.

The architecture of your repository should follow the following format:

```
$pwd
/home/nicolas/c-environment
$tree --charset=ascii
.
|-- build_program.sh
|-- clean_repository.sh
|-- main.c
`-- setup.sh

0 directories, 4 files
```

4.2 A word on cheating

You have to remember that you should be studying for your own good. Cheating will not bring you any good in the long term; it is fine not to be able to finish every exercise of the subject, your main goal is to train and learn things.

Any form of cheating will immediately bring your grade down to 0. Additionally, your main teacher will be taken notice of that.

5 Exercises

In between each level, you are asked to commit your work with git. **If you do not commit your changes between each level, you will loose points**.

5.1 Level 1: A simple program

In `main.c`, write down a program that prints all numbers from 1 to 10. After each number should be displayed a newline. The return code of the program should be 0[1].

```
$./print_numbers
1
2
3
4
5
6
7
8
9
10
```

For this level only, to test your program, you can use a third-party platform like **this one**. To pass this level, file `main.c` must be pushed to your repository and must contain the code of the program.

5.2 Level 2: Compilation!

The goal for this level is to be able to compile the program you just wrote natively (i.e. without a third-party platform).

You are expected to add two scripts to your repository to pass this level:

- `setup.sh`
- `build_program.sh`

The first one should install all of the softwares needed to compile your program on a fresh Ubuntu system. The second one should produce a binary that prints all numbers from 1 to 10, each followed by a newline. The return code of the program should be 0. The filename of the program does not matter.

Note: the binary that you generate is not to be commited!

```
$ls
build_program.sh  main.c  setup.sh
$./setup.sh
```

[1] This is controlled by the `return` statement of the `main` function. At any time in your terminal, enter `echo $?` to check out he return code of the last command executed

```
$./build_program.sh
$ls
a.out  build_program.sh  main.c  setup.sh
$./a.out
1
2
3
4
5
6
7
8
9
10
$ls
a.out  build_program.sh  main.c  setup.sh
```

You can use either `gcc` or `clang` to compile your code. Refer to their respective man pages for reference usage. Feel free to lookup tutorials online (e.g. **this one** for gcc).

5.3 Level 3: Cleanup script

To pass this exercise, write a cleanup script that removes all files generated by your build script and incorporate it to your repository. At this point, since the compilation script only creates one file, only this one should be removed by that script. If there is no file to delete, nothing should happen[2].

```
$ls
build_program.sh  clean_repository.sh  main.c  setup.sh
$./clean_repository.sh
$ ls
build_program.sh  clean_repository.sh  main.c  setup.sh
$./build_program.sh
$./a.out
1
2
3
4
5
```

[2]One can use the `test -f` statement to check if a file exists: **https://linuxize.com/post/bash-check-if-file-exists/#check-if-file-exists**

```
6
7
8
9
10
$ ls
a.out  build_program.sh  clean_repository.sh  main.c  setup.sh
$./clean_repository.sh
$ls
build_program.sh  clean_repository.sh  main.c  setup.sh
$./clean_repository.sh
$ls
build_program.sh  clean_repository.sh  main.c  setup.sh
```

5.4 Level 4: Name your program

To pass this exercise, your `build_program.sh` script should name your program `enumerate_numbers`. Your cleanup script should be adapted accordingly.

```
$ls
build_program.sh  clean_repository.sh  main.c  setup.sh
$./build_program.sh
$ls
build_program.sh  clean_repository.sh  enumerate_numbers  main.c
   setup.sh
$./enumerate_numbers
1
2
3
4
5
6
7
8
9
10
$./clean_repository.sh
$ls
build_program.sh  clean_repository.sh  main.c  setup.sh
```

5.5 Level 5: Guessing game

Now that you have a naive program, but most importantly, a working environment, you will write a slightly more useful program. To pass this level, you need to:

- Change your script such that your program is named `guessing_game` instead of `enumerate_numbers`
- Replace the code of your program such that, instead of enumerating numbers, it implements a guesing game, similar to the one from tutorial-1. Here are the specifications of the game:
  - On program startup, "I have in mind a number in between 1 and 100, can you find it?", with a line feed, should be displayed. A number to guess should be taken, at random, between 1 and 100
  - Then, the program should wait for user input
  - If the user inputs a number between 1 and 100, it should be compared to the random number taken. If the user number is bigger than the random number, "The number to guess is lower" should be displayed, with a line feed. If it is lower, "The number to guess is higher" should be displayed, with a line feed. In both cases, the program should wait for user input and repeat that step until the right number is given
  - If the right number is given, "You just found the number, it was indeed " should be displayed, followed by the number and a line feed. Then, the program should exit with an exit code of 0
  - At any time, if the user inputs something wrong (e.g. letters, a number greater than 100 or lower than 1), the program should exit with an exit code of 1 and an appropriate error message

Hint: `man scanf`, `man atoi`.

```
$ls
build_program.sh  clean_repository.sh  main.c  setup.sh
$./build_program.sh
$ls
build_program.sh  clean_repository.sh  guessing_game  main.c  setup.
   sh
$./guessing_game
I have in mind a number in between 1 and 100, can you find it?
50
The number to guess is higher
75
The number to guess is lower
63
The number to guess is lower
57
The number to guess is lower
```

```
53
The number to guess is higher
55
The number to guess is higher
56
You just found the number, it was indeed 56
$
```

5.6 Level 6: Command line arguments

The standard way to alter a program is to give it positional arguments. We have seen it plenty of times with basics UNIX commands. For example, when using `ls -l`, we invoke `ls` with a `-l` argument. Of course then, it is possible to retrieve arguments in your C code. The parameters to your main function, `argc` and `argv` provide access to the arguments provided by the user.

To pass this level, your program should:

- If invoked without argument (`./guessing_game`), behave like it used to do, which is to run the guessing game
- If invoked with an argument that is a number, run the guessing game between 1 and that number instead of in between 1 and 100 (e.g. `./guessing_game 50` should run the guessing game from 1 to 50)
- If invoked with an argument that is a number lower or equal to 1, display an appropriate error message and exit with a code 2
- If invoked with an argument that is not a number, display an appropriate error message and exit with a code 2
- If invoked with more than one argument, only consider the first one

Hint: `man atoi`.

```
$./guessing_game 33
I have in mind a number in between 1 and 33, can you find it?
16
The number to guess is higher
25
The number to guess is lower
20
The number to guess is lower
18
You just found the number, it was indeed 18
$
```

```
$./guessing_game yo
Invalid argument
$./guessing_game 0
Invalid argument
$./guessing_game -1
Invalid argument
$./guessing_game 5 yo
I have in mind a number in between 1 and 5, can you find it?
3
The number to guess is higher
4
You just found the number, it was indeed 4
$
```

5.7 Level 7: Makefile (bonus)

The standard way to write compilation and cleanup steps for a C project is to use the command `make`. It is most of the times working in pair with a `Makefile` file. Makefiles contain logic rules to apply: which compiler to use for a given rule, custom steps for cross-platforms compilation, specific workflows to only compile some parts of a codebase, etc.

To pass this level, replace `build_program.sh` and `clean_repository.sh` by a Makefile. Define a `build` rule to replace the first script and a `clean` rule to replace the other one.

Feel free to check out the man page for `make` and online resources. Since our program is simple, the resulting Makefile should be very slim.