University of Reading
Department of Computer Science
Pat Parslow, Rhian Taylor

Exercise Week 4
CS1PC20 / Autumn 2021
120 Minutes
Discussion date: Week 5

---

1. The exercises are formative – they are there to learn – but also assessed as part of your portfolio of work submitted for course works.
2. Exercises will be discussed on the Week indicated on the header. Additional (non-discussed) training tasks may be provided, and will be shown in red.
3. Make sure to schedule some time each week to keep up with the progress. It is not necessary to complete every task perfectly. If you are struggling, then reach out for help on Teams.

Exercise Introduction

Remember:

You should have done exercises 1 (bash) and 2 (git) and 3 (programming) before this exercise – we are creating a real program!

If you are finding you do not get the expected results, it is almost always (except where otherwise noted) because of a typing mistake or failure to follow the steps in order. Do check!

Be careful to distinguish between characters that look similar:

1 is not l is not I is not | for instance (these are one, lower case L, capital i and the pipe symbol, respectively)
O is not 0
~ is not – and ~ is called the "tilde"
_ is called underscore
# is called an 'octothorpe' but almost nobody actually calls it that, it is 'hash'
" is (on UK keyboards) shift 2. If you are cheating and using copy/paste, you may end up with a 'smart quote' which is not the right character. Do not use copy/paste.

Contents

Task 4: Programming Exercises (120 min) 2

Task 4: Programming Exercises (120 min)

You may want to view the video of me going through the exercise commands before you do it – I was not recording the outcomes, so it is done more quickly than I would expect you to, but it is still an hour long as I explain many of the features as I go along. You may get some additional hints about what the commands do from the video!

We don't expect for you to be writing long and complex C programs straight away. We also appreciate that moving to a CLI (Command Line Interface), while powerful, can be challenging especially when you are accustomed to interacting with primarily graphical interfaces on Operating Systems like Windows, and that C can be a lot less forgiving than that other snake-related language you may have used before (python).

Below we have included several steps for creating simple C programs of varying complexity that also cover more advanced concepts.

We want you to *experiment with them*, try running them and observe the result! Think about what you expect their output to be before you run them.

See what happens if you change the code (program) slightly – does it have the effect you expect?

Make your own changes to the program code. Use this as an opportunity to learn about C and programming practice.

We would like you to produce a markdown text file providing a short descriptions of your findings, i.e. all you need to provide is a brief description of what the command does (if possible write what you expected) and a short explanation of how it works where appropriate. Marks do not depend on you having the right 'prediction' – so avoid the temptation to run it and then record what happened as what you think will happen!

You should have completed the week 1 & 2 & 3 exercises first. Programming courses often start with "Hello World" (we did, in fact, in week 1) written as a 'main' function in C. We want to avoid that, and write our programs in a way that makes it easy to test them, so we will be writing functions in libraries, and using those libraries in a test program.

This week we will be writing our own testing framework, and exploring the "make" utility.

You do not need to type in the $ at the beginning of each line (in fact, you need to not type it in…)

Where the instructions say to edit a file to say something, you can use nano, vim, sed, echo commands… you'll probably want to stick with nano or vim though! If you are using nano, the commands for saving and exiting are listed along the bottom of the screen (ctrl-O to write the file, ctrl-X to exit nano). If you use vim (and git may put you in to vim if you are not using our standard virtual machine), you need to know it has two "modes" – command and insert mode. To enter insert mode, press "i" (or a, or o… or actually a number of others – i is for insert, a for append, o inserts a new line below the cursor and enters insert mode on it). To get in to command mode you press the Esc (escape) key. To save and quit, when you are in command mode, use ":wq"

1. `$ cd ~`

Do make sure you have the proper folder structure, and that your portfolio folder is a git repository.

If you have a folder called cs1pc20_portfolio instead of one called portfolio, you can rename it:

```
mv cs1pc20_portfolio portfolio
```

Portfolio should have week1, week2 and week3 folders in it, depending on which branch your git repository is currently on, and using

```
git status
```

should ***not*** say "fatal: not a git repository" if you are in a git repository. Assuming you have done exercises 2 and 3 you can recover your work from https://csgitlab.reading.ac.uk if necessary (see the end of Exercise 2 for the git clone command)

If your virtual machine is running out of disk space, follow the instructions on Blackboard to increase the disk size and the partitions.

1. `$ cd portfolio`
2. `$ git switch master`
3. `$ mkdir -p week4/framework`
4. `$ cd week4/framework`
5. `$ git branch framework`
6. `$ git switch framework`

Let's set up a folder structure to keep things neat and tidy.

And, because we do not like having to type things like the compile line, with its switches all the time, let us set up a Makefile.

"make" reads a Makefile and interprets a set of rules written in it, which tell the computer how to build the program we are working on. We will use a simple version, without the complexity of pattern matching.

What we want is a "make" goal (command) called "feature" which will set up our folder structure for us. "make" is picky about whitespace – the steps which define how to achieve a goal have to be on lines which start with a "tab". On most keyboards, tab is the key to the left of the Q key. In the exercise, this is shown as <tab>

Use nano to create a file called Makefile:

7. `$ nano Makefile`

The file contents should be:

```
feature:
<tab>mkdir $(NAME) ;\
<tab>cd $(NAME) && \
<tab>mkdir bin doc src test lib config ;\
<tab>echo "*" > bin/.gitignore ;\
<tab>echo "*" > lib/.gitignore
```

Because it is too easy to get the whitespace wrong, let's have a look at how to check it is right.
The 'cat' command can take options, and we will use -v to show Tabs and End-of-line :

8. `$ cat -vTE Makefile`

It should look like this:

```
feature:$
^Imkdir $(NAME) ;\$
^Icd $(NAME) && \$
^Imkdir bin doc src test lib config$
^Iecho "*" > bin/.gitignore ;\$
^Iecho "*" > lib/.gitignore
```

The <tab> characters are shown as `^I` and the end-of-line characters as `$`

If the output from `cat -vTE Makefile` does not look like this, go back in to nano and correct it.

9. `$ make feature NAME=test_output`
10. `$ ls -al test_output`

You should see a list of the folders within the test_output folder.

11. `$ git add Makefile`
12. `$ git commit -m "Setting up Makefile to create feature folders"`
13. `$ git push`

Now we are going to write a simple test framework. It involves reading and writing files, loops (while loops), if statements (control flow) and some string handling, as well as the idea of arrays (strings are 'arrays' of characters).

It also, unavoidably, contains pointers (see the *fp and *tests variables in the code). Do not worry unduly about these at the moment – it is jut a way of passing arrays to functions, which we will cover later.

14. $ cd test_output; `cd src`
15. Create a file called test_outputs.c with the following contents:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

/** define some constant values for size of data
* noting of course that if your data needs bigger values, you have to
* edit the source code and change the constants defined here
*/
#define COM_SIZ 60
#define ARG_SIZ 1024
#define RES_SIZ 1024

/** \fn int main(int argc, char *argv[])

    This test program calls an existing executable and checks the
    outputs to standard output meet the expected values.
    It should be called with:
    test_outputs <filename which contains test definitions>

*/

int main(int argc, char *argv[]) {
  FILE *fp;       /**< fp is a pointer used to give access to the file
descriptor of the pipe */
  FILE *tests;
  char command[COM_SIZ];
  char arguments[ARG_SIZ];
  char expected[RES_SIZ];
  char actual[RES_SIZ];
  char command_line[COM_SIZ + ARG_SIZ];

  /** try to open the file named on the command line */
  tests=fopen(argv[1], "rt");
  if (tests==NULL) {
    printf("Error  opening file %s\n",argv[1]);
    return 1;
  }

  /** we will read each line from  the file.
  * These should be structured as:
  * command to run
  * inputs
  * expected output

  * Note: this could go horribly wrong if the input file is not
  * properly formatted
  */
  while (fgets(command, COM_SIZ, tests) != NULL) {
    fgets(arguments, ARG_SIZ, tests);
    fgets(expected, RES_SIZ, tests);

    /** string handling in C can be cumbersome.
    * typically suggestions online make use of "malloc" and
    * "strcpy" and "strcat"
    * but these complicate things and are arguably not good practice
    * strtok gives us a useful shortcut to
    * remove newlines (the way it is used here)
    */
    strtok(command, "\n");
    snprintf(command_line, sizeof command_line, "%s %s", command, arguments);
  /** Now we call the command, with the arguments and capture the result
  * so we can compare it to the expected result.
  * the "popen" command opens a special type of 'file' called a 'pipe'
  */

    fp= popen(command_line , "r");
    if (fp == NULL){
      printf("Failed to run command\n");
      exit(1);
    }
  /** This is how we get the result back out of the pipe we opened
  * after reading the result in to "actual" we compare it to the expected value
  * strcmp is slightly unusual - it returns
  * 0 if the strings are the same,
  * >0 if string1 is
  * bigger than string2, and
  * <0 if string1 is less than string2
  * because 0 is 'false', we negate (!) the result to test if
  * they are the same.
  */
    char message[RES_SIZ + RES_SIZ + 21];

    while(fgets(actual, sizeof(actual), fp) != NULL) {
      /** we create a message to let us know what was expected and what we got
      * note that we have split the line in the next statement - that is
      * fine, we can!
      */
      snprintf(message, sizeof message, "%s %s %s %s",
         "Expected ", expected, " and got ", actual);
      printf("%s",message);
      /** Because we want the test suite to keep running, we use an if
      * statement rather than the assert function
      */
      if(!strcmp(actual, expected)) {
        printf("OK\n");
      }
      else {
        printf("FAILED\n");
      }
    }
    /** if we don't close file handles, we risk using up the machines resources
    */
    pclose(fp);
  }
  fclose(tests);
  }
  return 0;
}
```

OK that is big, and a lot of typing. When you compile it in the next step, though, you should not get any errors or warnings, so if you do, look at what they say and fix the problems if you can – or ask for help!

16. `$ gcc -Wall -pedantic test_outputs.c -o test_outputs`

Note that we have written this as a main program, and compile it straight to an executable, rather than creating library code.

If it has compiled without any warnings or errors, the next two commands *should show errors*

17. `$ ./test_outputs file_does_not_exist`

18. `$ ./test_outputs`

19. Create a file called op_test with the following contents (note that there is no "newline" at the end of the file):

```
wc -l
test_outputs.c
108 test_outputs.c
wc -l
test_outputs.c
82 test_outputs.c
wc -l
test_outputs.c
108 Test_outputs.c
```

When you run the test_outputs program you have just typed in and compiled with this as an input file, what do you expect to happen?

```
20. $./test_outputs op_test
21. $git add test_outputs.c
22. $git add op_test
23. $git commit -m "test framework and sample test suite"
24. $git push
```

Hints

- We have used fixed size arrays – we will look at more flexible options later
- We have used very minimal sets of tests – think about what other values we should test
- We have not handled what happens if the input file is badly formatted (e.g. not enough lines, a spare blank line on the end…). Why might that be a problem for us?
- The "while" loop lets us *loop* for as long as the condition given is true.
- You can use the *history* shell command to see what you have done.
- You can use

```
$history > my_work_log.md
```

To produce a file with what you have done so that you can edit it with comments.

Portfolio (directory: portfolio/week4/make_and_test.md)

portfolio/week4/make_and_test.md	
	An enumerated list that provides for each command a prose sentence what the commands did. Where your expectation differs from observed behaviour, include a sentence describing what surprised you. See if you can explain what the commands did (briefly!) You definitely do not need to reproduce the code in your .md file!