

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) before this exercise – we are creating a real program!

## Contents

Task 3: Programming Exercises (120 min)

2

## Task 3: 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, so the video is only 20 minutes long. 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 a simple C program 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.** Perhaps 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 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.

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!

1. `$ cd ~`
2. `$ cd portfolio`
3. `$ mkdir week3`
4. `$ mkdir week3/greeting`
5. `$ cd week3/greeting`
  
6. `$ git branch greeting`
7. `$ git switch greeting`
8. Create a file called `greeting.c` with the following contents:

```
#include <stdio.h>
int greet(void) {
    printf("Hello world!\n");
    return 0;
}
```

9. `$ gcc -Wall -pedantic -c greeting.c -o greeting.o`
10. Create a file called `test_result.c` with the following contents:

```
#include <assert.h>
#include "greeting.h"

int main(void) {
    assert(0==greet());
    return 0;
}
```

11. And create a file called `greeting.h` with the following contents:

```
int greet(void);
```

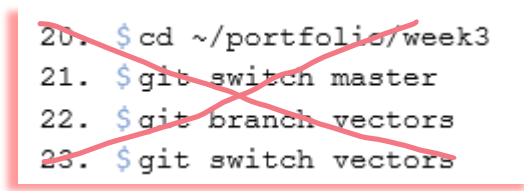
```
12. $ echo greeting.o >> ~/portfolio/.gitignore
13. $ echo libgreet.a >> ~/portfolio/.gitignore
14. $ ar rv libgreet.a greeting.o
15. $ gcc test_result.c -o test1 -L. -lgreet -I.
16. $ ./test1
17. $ git add -A
18. $ git commit -m "greeting library and greeting test program"
19. $ git push
```

OK that's a definite "pause" moment – what is all this "ar" stuff, and all those flags on the gcc in line 13?

Do the math!

We will come back to look at testing, and what all the flags mean and so on later. For now let's write some other programs.

```
20. $ git switch master
21. $ git branch vectors
22. $ git switch vectors
23. $ cd ~/portfolio/week3
24. $ mkdir vectors
25. $ cd vectors
```



```
20. $ cd ~/portfolio/week3
21. $ git switch master
22. $ git branch vectors
23. $ git switch vectors
```

*Figure 1 Used to say this - why would this be wrong?*

26. Create a file called `vector.h` with the following contents:

```
#define SI2 3
int add_vectors(int x[], int y[], int z[]);
```

27. Create a file called `test_vector_add.c` with the following contents:

```
#include <assert.h>
#include "vector.h"

/** A simple test framework for vector library
 * we will be improving on this later
 */
int main(void) {
    /** xvec and yvec will be inputs to our vector arithmetic routines
```

```

* zvec will take the return value
*/
int xvec[SIZ]={1,2,3};
int yvec[SIZ]={5,0,2};
int zvec[SIZ];
add_vectors(xvec,yvec,zvec);
/** We want to check each element of the returned vector
*/
assert(6==zvec[0]);
assert(2==zvec[1]);
assert(5==zvec[2]);
/** If the asserts worked, there wasn't an error so return 0
*/
return 0;
}

```

28. And now create the code to actually “do the math” – vector.c

```

#include "vector.h"

/** A simple fixed size vector addition routine
* Add each element of x to corresponding element of y, storing answer in z
* It is the calling codes responsibility to ensure they are the right size
* and that they have been declared.
* We return an error code (0 in this case showing no error), but will add the
* program logic to handle actual errors later
*/
int add_vectors(int x[], int y[], int z[]) {

    for (int i=0;i<SIZ;i++)
        z[i]=x[i]+y[i];
    return 0;
}

```

Like we did with the greeting code, we will compile to a library, and then use that library when we compile the test program:

```

29. $gcc -Wall -pedantic -c vector.c -o vector.o
30. $ar rv libvector.a vector.o
31. $gcc test_vector_add.c -o test_vector_add1 -L. -lvector -I.
32. $./test_vector_add1
33. $git add -A
34. $git commit -m "code to add two vectors of fixed size"
35. $git push

```

Well – that isn’t very clear, but if it says nothing, it worked!

36. Change the `assert(5==zvec[2]);` line to be `assert(5==zvec[1]);` and recompile test to see what happens.

Now let us add a function to calculate the “dot product” of two fixed size vectors.

37. Edit vector.h so it contains this:

```
#define SIZ 3
int add_vectors(int x[], int y[], int z[]);
int dot_product(int x[], int y[], int z[]);
```

38. Edit vector.c so it contains this:

```
#include "vector.h"

/** A simple fixed size vector addition routine
 * Add each element of x to corresponding element of y, storing answer in z
 * It is the calling codes responsibility to ensure they are the right size
 * and that they have been declared.
 * We return an error code (0 in this case showing no error), but will add the
 * program logic to handle actual errors later
 */
int add_vectors(int x[], int y[], int z[]) {

    for (int i=0;i<SIZ;i++)
        z[i]=x[i]+y[i];
    return 0;
}

/** A simple fixed size dot product routine
 * multiply each element of x to corresponding element of y, adding up totals
 * It is the calling codes responsibility to ensure they are the right size
 * and that they have been declared.
 * We return the actual value we have calculated
 * We may need program logic to handle actual errors later
 */
int dot_product(int x[], int y[]) {
    /** res <- a local variable to hold the result as we calculate it
    */
    int res = 0;
    for (int i=0;i<SIZ;i++)
        res=res + x[i]*y[i];
    return res;
}
```

39. And create test\_vector\_dot\_product.c so it contains:

```
#include <assert.h>
#include "vector.h"

/** A simple test framework for vector library
 */
int main(void) {
    /** xvec and yvec will be inputs to our vector arithmetic routines
    */
    int xvec[SIZ]={1,2,3};
    int yvec[SIZ]={5,0,2};
    int result;
    result=dot_product(xvec,yvec);
    /** We want to check each element of the returned vector
    */
    assert(11==result);
    return 0;
}
```

---

And then we compile to a library, and use that library when we compile the new test program:

```
1. $ gcc -Wall -pedantic -c vector.c -o vector.o
2. $ ar rv libvector.a vector.o
3. $ gcc test_vector_dot_product.c -o test_vector_dot_product1 -L. -lvector -I.
4. $ ./test_vector_dot_product1
5. $ git add -A
6. $ git commit -m "code to calculate dot product of two vectors of fixed size"
7. $ 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 introduced functions which return error codes, and ones which return calculated values. These use the same mechanism. Why might that be a problem for us?
- The “for” loop lets us *iterate* through a set of values, but the syntax (how it is written) might be confusing – this will be explained in more detail later.
- The *dot product* is a function commonly used in neural networks and a whole range of other applications – the maths really is not all that scary!
- Consider what errors may occur, and write tests to catch them – what if the numbers passed in to our vector functions were very big?
- 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/week3/c-programs.md)

portfolio/week3/c-programs.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!)