

# Week 02 - Workshop

August 9, 2020

## 1 Worksheet 1

### 1.1 Overview

The workshop for Week 2 will start with a mini-tutorial on few essential tools to succeed in COMP20003:

1. A few bash commands
2. UNIX tools to compile your program
3. UNIX tools to debug your code

We are going to use the GNU compiler gcc, another GNU tool that will make your life easier to compile big programs, Make, and gdb, the command line debugger.

This year we're using a cloud notebook server, this is managed mostly by us, but as a forewarning there may be some hiccups along the way. If you're interested in running the notebooks locally, we'll happily offer support for you doing so using Docker, the experience should be very similar. The JupyterLab interface (what you're very probably seeing here) also allows you to open a terminal, this will become necessary for assignments and debugging your work.

The rest of the workshop you will intend to refresh your C programming skills and get a brief taste of some of the tools you'll be expected to use heavily in your assignment work. For simplicity, this year your assignment code will only be required to run correctly on the notebook servers (here).

### 1.2 Programming Exercises

The main task for this week is to ensure that you're familiar with the tools we'll be use in the subject and are able to use and understand a number of simple unix commands.

If you feel some of the programs below are too trivial for your current proficiency in C, feel free to skip ahead.

**Programming 1.1** If you haven't actually written and compiled a program in Unix before, or might be a little rusty, start with compiling the traditional "Hello, world!" program. This program simply prints "Hello, world!" to stdout (stdout is the screen, unless redirected). You can make a subdirectory for your work. The following commands may be useful: - ls (*list*) - cd (*change directory*) - mkdir (*make directory*) - cat (*concatenate*) - more (*show the file, one screen at a time*) - nano (*file (open the nano text editor on the given file)*) - man *command/function* (*See the full explanation of how to use the given command or function*)

Particularly the *man* program, which lets you see how each program works (e.g. *man mkdir* will show you how to use *mkdir*), you can type 'q' to exit the man pages.

Use an editor of your choice to write a file called *hello.c*. Note that when you are connected to the Unix machines and you save a file in the MobaXterm editor, it will automatically upload it to your directory on the Unix machines. Check that the file is where you think it should be (the *ls* command gives you a listing of the directory contents and you can read the file using the *less* command, i.e. *less hello.c*).

When you are ready to compile, *gcc hello.c* will create the executable *a.out*. just type *./a.out* in the Unix window to run the program. If you can't stand the name *a.out*, or the fact that all your executables will be called *a.out*, thus overwriting each other if they are in the same directory, run *gcc -o hello hello.c*, the *-o* flag will name the output flag whatever you put after it (*hello* in this case). Just be a bit careful what you call your executables; for example if you call it "test" and then try to run it, you might get a surprise (Type *which test* to try to work out what's happening, or *man test*. In order to run your executable you'll need to type *./test*)

```
[ ]: #include <stdio.h>

int main(int argc, char **argv){
    printf("Hello, world!\n");
    return 0;
}
```

**Programming 1.2** In this part, we'll ask you to write a number of small simple programs.

*Problem 1:* Define a static array *A* of size 7

```
[ ]: /*%test_script:week2/test_array.sh
#include <stdio.h>

/* Define an I, which will be the size of the array, A. */

int main(int argc, char **argv){
    /* Define an array, A, of integers */
    /* write a loop which populates A with multiples of 12 */
    /* print out A[0], A[2] and A[5] */

    return 0;
}
```

*Problem 2:* Take your program from Problem 1 and instead use random numbers. You may find *rand* from *stdlib.h* is useful for this. Consider using *man rand* to find out what this function does and how to use it.

```
[ ]: #include <stdlib.h>
/* Your code below here */
```

*Problem 3:* For the next problem, try utilising *scanf* or *getline* from *stdio.h* to take in integers from *stdin* (the keyboard). As usual, you can find more about how these functions can be used and

exactly what they do using *man scanf* or similar. As a note, the below code snippet in the notebook uses some non-standard syntax to simulate some user interaction which is not sensibly possible through the notebook format, you'll likely encounter more of this as we go on. Feel free to ignore that and instead try this on the department machines.

```
[ ]: ///%test_script:week2/test_array_storage.sh
///%stdin: "1 2 3 4 5 6 7 "
/* Your code from Problem 2 here, utilising scanf instead of rand. */
```

*Problem 4:* Make this array hold strings instead, store two characters at a time to begin with, you can hardcode the space for this.

```
[ ]: ///%test_script:week2/test_string_storage.sh
///%stdin: "1a2b3c4d5e6f7g"
/* Your code from Problem 3 here, storing the values as strings instead. */
```

### 1.2.1 Free and Malloc

For the next problem, we introduce the **malloc** and **free** functions briefly. We'll explore more about these next week.

**malloc** is simply short for memory allocate. You can check *man malloc* to see how it works. Of note are a few things:

```
#include <stdlib.h>
```

We need to include `stdlib.h` to use `malloc`.

```
void *malloc(size_t size);
```

`malloc` takes a single argument, the number of bytes to allocate space for. Different data types (`int`, `char`, `pointer`, `double`) have different sizes, you can use **sizeof** to get the size of these values. (e.g. `sizeof(int)` will give you the number of bytes needed to store one `int` value). This function returns a pointer to the space allocated using `malloc`.

**free** is the counterpart to `malloc`, every bit of space you `malloc` should eventually be freed. All there is to it is to put the address you get from the `malloc` back in as the argument to `free`.

Here is a small example problem to show how to use these.

```
[ ]: #include <stdio.h>
// Include the header we need for malloc.
#include <stdlib.h>
// malloc will return NULL if it fails, asserts are a quick way to check success.
#include <assert.h>

#define NUMBEROFINTS 2

int main(int argc, char **argv){
    // We declare space to store a pointer of type int.
    // The value NULL is just an alias for 0 (of pointer type).
```

```

int *values = NULL;

// Here we allocate space for two integers. The result is a generic pointer,
// so we also set the type to a pointer to integers.
values = (int *) malloc(sizeof(int) * NUMBEROFINTS);
// Check memory was successfully allocated
assert(values != NULL);

// Set each of the integers.
values[0] = 10;
values[1] = 32;

// Print the values out.
printf("%d %d\n", values[0], values[1]);

// free the malloced memory.
free(values);

return 0;
}

```

**Memory Errors** In the above script, try each of these in the above code and see what error message you get to get familiar with common issues: - Comment out the malloc line (and the assert), - Comment out the free line, - Comment out the line where you set values[0], - Set a value for values[2], - Change the value of NUMBEROFINTS to 1.

*Problem 5:* Make the array instead hold strings of variable length. For this, take first a number, which will be the number of characters stored in each string in the array, and then read that many characters into that array. As usual, print out A[0], A[2] and A[5]. The notebook code again utilises some information to provide more useful hints if you've made a mistake. Note also that the Notebook interpreter will behave a little differently to your regular gcc invocation in this case as a number of additional compilation options are turned on to detect issues with what you have written.

*Note:* You will also have to free the memory you have allocated after you are done when running in the notebook.

```

[ ]: %%test_script:week2/test_variable_string_storage.sh
%%stdin: "1a2ba3cab4dabc5eabcd6fabcd7gabcdef"
%%stdout: "acabfabcd"
/* Your code from Problem 4 here, utilising malloc to allocate exactly the
→ amount of space you need for each string. */

```

*Problem 6:* This involves something that we won't formally cover until next week's workshop, but if you've seen it before or want to give it a go anyway, please do! Try declaring a linked list and using it in place of the array. In this case, print out the first three items after all seven items have been added to the list.

```
[ ]: ///test_script:week2/test_LL_string.sh
///stdin: "1a2ba3cab4dabc5eabcd6fabcd7gabcdef"
///stdout: "gabcdeffabcdeeeabcd"
```

Every week we'll have a few programming challenges, they're intended to be fairly simple but aren't necessarily easy problems to solve. Some of these are designed to force you to confront the simplicity of C, while others are intended to be fun challenges which you can use the algorithms and data structures we've learned in the course to solve.

**Programming 1.3** In this problem, you'll be asked to find all the prime numbers from 1 to 1000. Prime numbers are used in all kinds of circumstances, particularly in fields such as cryptography, hashing among many others. Any method will be sufficient for this problem. The Sieve of Eratosthenes is one algorithm which you can try implementing, but there are plenty of others. A prime is prime if it is not the product of any lesser natural numbers except for 1 and itself, for example, 1, 2 and 3 should be prime by this definition, but 4, being the product of 2 and 2, is not.

```
[ ]: ///test_script:week2/test_prime.sh
#include <stdio.h>
#include <stdlib.h>
#define PRIMECOUNT 1000

int main(int argc, char **argv){
    /* The primes 1 - 1000, if n is prime,
       primes[n - 1] == 1 and 0 otherwise. */
    int primes[PRIMECOUNT];
    int i;

    for(i = 0; i < PRIMECOUNT; i++){
        primes[i] = 0;
    }

    /* Write a prime checking algorithm here. */

    /* ----- */

    printf("All primes found from 1 - 1000:\n");
    for(i = 0; i < PRIMECOUNT; i++){
        if(primes[i]){
            printf("%d ", i + 1);
        }
    }
    printf("\n");

    return 0;
}
```

**Programming Challenge 1.1** Prime numbers are fairly useful on their own, but another fun challenge is to find what are known as emirp primes. Emirp primes are primes which are *also* primes

when reversed. For this challenge, you need to print out the list of all primes that are also primes (as defined in the challenge above) when reversed. [1]

```
[ ]: ///test_script:week2/test_emirp.sh  
/* Just copy your code from above and add the emirp prime check. */
```

**Programming Challenge 1.2** As an extension to Programming Challenge 1.1, take a value from standard input and show the emirp primes up to that value. For this challenge you will have to malloc space.

```
[ ]: ///stdin: "1000"
```

**Programming Challenge 1.3** As an extension to Programming Challenge 1.2, instead of returning the emirp primes up to the given value, return the first emirp primes. e.g. if the number sent on stdin is 5, your program should print "1 2 3 5 7", and if the number is 8, your program should print "1 2 3 5 7 11 13 17".

Though there is a function to do a lot of this work for you that we'll meet next week, for this task try to perform this task using only malloc and free. 1. Use malloc to allocate new space if you need to look at more primes. 2. Copy over the existing data to this new space and fill in the primes following. 3. Free the old space.

```
[ ]: ///stdin: "57"
```

### 1.2.2 GDB and Valgrind

We'll revisit some of the other functionality of GDB and Valgrind later in week 4 where we examine other difficult problems which you might not have had to solve in the past. But this week now we'll be looking at how debugging can help solve some of the issues which some careful inspection might allow you to solve. A number of these issues can be detected by the notebook compilation, but the notebook is not necessarily that helpful in working out how to solve the issue.

For this exercise, we will get you to use **GDB** and **Valgrind**. The **GDB reference card** file will likely be useful, as well as the **Valgrind tutorial** on the LMS under Resources. Valgrind is a great tool to help find memory leaks, but can be used for plenty of other things too!

**NOTE:** You will get much more useful debugging information by compiling with `-g`, this adds additional debugging information which tends to be extremely important. Add this to all your compilations.

### 1.2.3 Quick summary of basic commands for GDB

To open your program using gdb, you can use: `gdb program_name` where *program\_name* is the name of your program.

Once you're inside gdb, you can use: `- b filename.c:lineNum` (to set a breakpoint at lineNum in filename.c) `- r` (to run your program, reminder: you can essentially treat `r/run` as your program's name, so additional arguments and input/output redirections can come after the `r` command) `- CTRL+c` (to stop your program executing) `- c` (continue) `- n` (next) `- s` (step) `- d #number` (disable breakpoint #number for now) `- p name_variable` (prints the value of the variable with the given

name) - bt (backtrace, shows the program stack, what functions were called to get to where you are currently) - q (quit, exits gdb) - start (essentially sets a temporary breakpoint at the main function, starts your program running and then stops it) - info locals (gives information about all local variables in the scope you're in) - finish (finish the function you're currently in) - command #number (set some gdb commands to run every time you hit breakpoint #number, end this list with end)

gdb has a huge suite of additional commands (including a Python interpreter, remote debugging, the ability to set variables to different values while running, time travel and functions), so you will probably find it useful to gradually transfer a lot of your usual debugging techniques over to gdb.

Once you've created a *breakpoint* and begun *running* your program, you can visualise your code.

Visualisation: - Ctrl-x followed by 1 (visualise the code you're debugging) - Ctrl-x followed by o (change focus of the screen from code to gdb terminal) - Ctrl-x followed by a (kill visualisation of code)

You can also find a number of issues using Valgrind.

### 1.2.4 Checking Memory in Valgrind:

You can use the following command to check for errors you've made in memory allocation using valgrind on the university servers. `valgrind -tool=memcheck program_name -tool=memcheck` specifies that valgrind should use the memory error checker, there are other tools, such as Cachegrind, Callgrind, Helgrind, DRD, Massif, DHAT, BBV, Lackey and (technically) Nulgrind. You can find what these do at [The Valgrind Manual](#).

*program\_name* is where you should place your program. Unlike gdb, you pass all your arguments when you call valgrind, so you would also add your arguments (and input redirections) here.

If you want a detailed description of memory leaks, add the option `-leak-check=yes` before your program's name (and arguments).

The program to debug (debug\_me.c) should be under the /week2/ folder (and if you run it here, you can see some interesting shortcomings of the error checking method used in the notebooks):

```
[ ]: //test_script:week2/test_fact.sh
#include <stdio.h>

int main()
{
    int i, num, j;
    printf ("Enter the number: ");
    scanf ("%d", &num );

    for (i=1; i<num; i++)
        j=j*i;

    printf("The factorial of %d is %d\n",num,j);
}
```

The malloc and free example is also given under the week2/ folder as malloc\_example.c, it would be worthwhile to try introducing each of the bugs we introduced earlier to see how valgrind tells you about them.

### 1.2.5 VGDB

Valgrind also offers a tool - vgdb, which can be used to quickly and decisively squish all the memory bugs valgrind can tell you about. A video tutorial on how to use this tool will be made available soon, so keep an eye out for that. It combines valgrind and gdb together, giving you the power of both in a way that greatly surpasses the power of either tool when used alone.

[1]: This challenge was mostly taken from [code-golf.io](https://code-golf.io), but there are some small differences in definition there. The challenge is essentially as difficult based on either definition, so feel free to use either.