Worksheet 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 are also trialling interactive notebooks for all workshops. These will allow you to run code directly in the worksheet itself. If it proves workable, these workbooks will include feedback on issues with the work you submit which may be difficult to otherwise discern (there's a lot you can do which works, but isn't what you want it to do). This is the first year we'll be trialling this, so please, if you find issues or points you think could be improved, please let us know on the LMS!

The subject involves work which will involve the use of heavier hitting tools, and to use those tools you'll need to be able to connect to the department machines. So, during the rest of the workshop you will make sure you can connect to the department machines and refresh your C programming skills. If you work from home, remember this week is a good time to set up the VPN so you can access the machines from outside the university [1].

Programming Exercises

The main task for this week is to check that you have access to the department computers nutmeg.eng.unimelb.edu.au and dimefox.eng.unimelb.edu.au, to make sure you can use MobaXterm (or other program of your choice) to ssh to the machines and to make sure you can use basic Unix command (as described in LMS → Workshops → Introduction to Unix and MobaXterm).

You will then refresh your C programming skills by writing a few small C programs, compiling and running them on the department machines. A few different programs are suggested below; you can skip those that you feel are too trivial for your current proficiency in C.

If you are unable to connect to the department machines, using the directories in LMS → Workshops → Introduction to Unix and MobaXterm, talk to your tutor immediately. Sometimes it is necessary to fix an account, in which case you'll need to see Student IT to correct the issue.
Remember that the servers (department machines) will need to be used to test the compilation/running of the assignments before submission. You may find tools such as MobaXTerm, FileZilla, Cyberduck or the humble scp command are useful to copy files to the servers (on UNIX-based systems, you can use scp -r for directories and man scp if you are unsure about the syntax of scp).

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). I suggest that you make a subdirectory under your home directory with the subject name, and then a further subdirectory named workshops. You can work in this directory, or you can create a further level in the hierarchy, with a subdirectory named week2 (see the Unix document for how to create and move around directories).

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*)

If you're using the notebook, you can see the results of your program here, but you will likely find being familiar with the server becomes very useful when it comes to assignments (and debugging), so you'll get the most value from running this program there.

```
#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. You may prefer to work either in the interactive notebook (in which you can use Ctrl+Enter to run the below code) or on the department machines.

Problem 1:

```
#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.

```
//%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.

```
//%stdin: "1a2b3c4d5e6f7g"
/* Your code from Problem 3 here, storing the values as strings instead. */
```

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.

```
//%stdin: "1a2ba3cab4dabc5eabcd6fabcde7gabcdef"
//%stdout: "acabfabcde"
//%memtotalnoterm: 28
//%memtotal: 35
//%memaux: 56
/* 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.

```
//%stdin: "1a2ba3cab4dabc5eabcd6fabcde7gabcdef"
//%stdout: "gabcdeffabcdeeabcd"
//%memexpect: 7 * sizeof(struct node) + 35
/* Declare your struct as "struct node" so memory expectations can be assessed */
```

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.

```
#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. [2]

```
/* Just copy your code from above and add the emirp prime check. */
```

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.

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 *run*ning 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.

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) can be found on the LMS under the Week 2 workshop on the LMS it's contents are as follows (and if you run it, you can see some interesting shortcomings of the error checking method used in the notebooks):

```
# 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);
}
```

[1]: VPN instructions: http://ask.unimelb.edu.au/app/answers/detail/a_id/6156

[2]: This challenge was mostly taken from 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.