Operating Systems

Practical 3

GNU Emacs
and
C Programming

Dr. Kevin Farrell

February 2025

Table of Contents

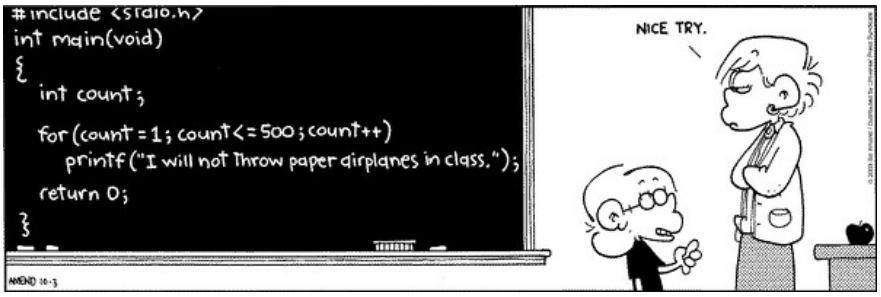
Legal

1. Introduction and Objectives

If you have not finished Practical 2, please do so before starting this one.

The following list is a summary of the tasks you will undertake in this Practical. The detailed steps are given later.

1. Install Emacs, a sophisticated text editor.
2. Briefly work through the Emacs editor guided tour and tutorial.
3. Learn about the layout/anatomy of a C program
4. Work through an online C programming Tutorial
5. Complete some exercises asking you to write some basic C Programs.



2. Installing GNU Emacs, and running its built-in Tutorial

2.1. First, boot your VM. Then, on VirtualBox, go to the **Machine** → **Settings** → **Storage**, and add the **Mageia-8-x86_64.iso** to the Optical Drive. This will be needed for installing packages later. If you are using VMware on Windows or UTM on a Mac, please ask your Lecturer for assistance for this step.

2.2. Now run the **Konqueror** Internet browser rather than Firefox, and open this handout from within your Mageia 8 Linux system.

2.3. Switch your VM to full-screen mode. On Windows hosts, you can generally use **CTRL-F** in VirtualBox to do this. On Apple Macs, and VMware on Windows, the key sequence is different – ask for help if you can't figure it out.

2.4. Then, go to the **Practicals** → **Supplementary Resources** section on the OS BrightSpace page, and download the "**OS Reflective Journal Template 2025**" file to your Linux system. Move it to your **Documents** directory, rename it to:

Firstname_Surname_Practical3_ReflectiveJournal.odt

and open it.

2.5. According to the GNU Emacs website (below), GNU Emacs, or simply Emacs, is *"an extensible, customizable text editor. It has content-sensitive editing modes, including syntax colouring (in C for example) and complete built-in documentation, including a tutorial for new users. It is highly customizable, and there are a large number of extensions available for it that add other functionality, including a project planner, debugger interface, compilation support"*. In this section, you're first going to get a brief overview of the editor, and then you're going to work through some elements of the built-in tutorial.

2.6. Start by having a brief look at the Emacs guided tour located at the URL below. Don't spend a huge amount of time on this. The idea is just to get an overview.

https://www.gnu.org/software/emacs/tour

2.7. Before you proceed, you need to make sure you have a working Internet connection. If you used your VM at home and you saved your machine state, your VM's network configuration will have the DNS server details assigned as the IP address of your Broadband Router. However, your Router is not accessible from outside your home. Therefore, if you are using your VM on campus, please follow these instructions to assign a valid DNS server to your VM: Right-click the World icon on the bottom right of the panel, and select "Disconnect Wired (Ethernet)". You should see a message that your network is connected. Then, right-click the same icon and select "Connect Wired (Ethernet)". This should update your network configuration with a valid DNS server.

2.8. Run a **Konsole** window, log in as root, and install the **Emacs** editor by typing the commands below. :

```
$ su -
# urpmi emacs
```

2.9. Answer "y" to the question about installing additional packages. Emacs and associated dependency packages should then install. If you get any errors, ask me for assistance. Once all packages are installed, log out as root, by typing "exit".

2.10. Start Emacs by accessing the **Development** menu from the menu button at the bottom Left-hand side of the icon panel. This would be a good time to add the Emacs application to the panel by right-clicking it and selecting **"Add to Panel (Widget)"**

2.11. **Do not** resize the Emacs window. If you did, close Emacs and run it again. This is because the various key strokes you are about to learn, only work properly with the default window size. Later, when you're using Emacs, you can choose whatever size of the window you wish. You can also set your choice of the default font.

2.12. Briefly, work through *some* of the Emacs tutorial, simply by clicking on **Emacs Tutorial** in the editor. You should aim to spend no more than about **15 minutes** on this. So work through it as quickly as you can. The aim is not to remember every little detail, but to get a sense of the various **key sequences** that the editor uses. **Note**: as stated above, DO NOT resize the Emacs window during the tutorial as this will change the layout of the text, which will then not match with the tutorial instructions.

2.13. Again, to emphasise, one of your objectives with Emacs is to learn the various key sequences. The key sequence to search for a string is:

`C-s`

where `C` stands for the CTRL key.

The key sequence to cancel the previous key sequence is:

`C-g`

where, again, the `C` stands for the CTRL key.

2.14. In the tutorial, see if you can find the key-sequences for the actions listed below. Write into your Reflective Journal on page 2 after the Table, what these key sequences are:

1. Find a file (also used to open a new file)
2. Save the file
3. Move to the end of a line
4. Move to the beginning of a line
5. Undo
6. Split the screen into two windows
7. Quit Emacs

3. Anatomy of a C Programming

3.1. Before we start writing some C programs, let's look at the layout of a C program – what we might call the 'anatomy' of a C program! **Note**: we often refer to our C (or indeed Java) code as "**Source Code**". This terms tells us it was written by a human; hence, the term "**Open Source**" for example, which indicates that the code is open and available for people to copy, use and modify.

3.2. **Note on comments in C**: The standard C comment is a set of symbols used as follows:

```
/* This is a comment */
```

3.3. Comments can also be written as follows:

```
/*
This is another comment
*/
```

3.4. However, we can also use C++/Java style commenting, which is often more convenient:

```
//This style of comment can be also used in modern C
```

3.5. The general layout of a C program is as follows (line numbers shown on the left):

```
1    #include <stdio.h>
2    #include "myheaderfile.h"
3
4    //Function prototypes not in header file go here;
5    //for eg:
6    void printMyAge(int age);
7
8    //Global variable declarations not in header
9    //file go here; for eg:
10   float globalY;
11
12   int main() {
13        //Local variable declarations go here; for eg:
14        int age;
15        //Variable definitions go here for both
16        //global and local variables
17        globalY = 3.14;
18        age = 21;
19
20        //Local variable declarations and definitions
21        //can be combined:
```

```
22        int age = 21; //alternative to line 14 + 18
23
24        //We can print something to the console:
25        printf("I will print my age!\n");
26
27        /* Calls to functions go here; for eg:
28        printmyAge(age);
29        return 0;
30    }
```

3.6. At lines 1 and 2 above, we see the inclusion of two header files. All C programs begin with the inclusion of ***header file(s)***. A header file contains the ***declarations*** of functions and/or variables. Note that in C++/Java, the term "method" is used instead of "function". But, essentially, they are the same thing. Header files always end in **".h"**

3.7. System header files are those which come with the C compiler and associated libraries (more on libraries later). To include a system header file, surround it with angled brackets. We can see this at Line 1 above, where we include the header file for Standard Input and Output, needed when we wish to do printing to the console:

```
1    #include <stdio.h>
```

3.8. Before a function (method) can be ***called*** (or invoked) in a C program, it must be ***declared***; that is, we must have a function ***prototype***: this declaration shows the data-type returned by the function along with the number and data-type of any parameters passed to the function. It does **not** contain the code itself of the function. That code is stored elsewhere – see later. There are two ways to include a function prototype: In the first way, we simply write the function prototype above **`main()`** as shown at Line 6:

```
6    void printMyAge(int age);
```

3.9. In the second way, we put the declarations of our own functions into a header file of our own. To include our own header file, surround it with double-quotes as shown in Line 2 above:

```
2    #include "myheaderfile.h"
```

3.10. ***Every*** C program must have a **`main()`** function, as shown on line 12 above:

```
12   int main() {
```

3.11. Notice the open curly bracket, and the final curly bracket at Line 30, just like in Java:

```
30   }
```

3.12. Note that sometimes we refer to the **`main()`** ***function*** as the **`main()`** ***program***.

3.13. Note also that all text after the open curly bracket is indented; i.e. moved towards the right. This makes our code readable.

3.14. Header files are also frequently used to store the declarations of ***global*** variables. These are variables that are accessible to all functions – they have ***"global scope"***; unlike local variables, which only have scope (can only be accessed) within the functions they are declared. We don't store local variables in header files.

3.15. Alternatively, global variables can be declared explicitly. They are always placed above **`main()`** as shown in Line 10:

```
10   float globalY;
```

3.16. Local variables are declared ***inside*** the **`main()`** function, as shown in Line 14.

```
14        int age;
```

3.17. We make a distinction between "declaration" and "definition". Declaring a variable simply gives the name of the variable preceded by its data-type. Defining a variable gives it a value. We can define both global and local variables inside **`main()`**, as shown in Lines 17 and 18:

```
17        globalY = 3.14;
18        age = 21;
```

3.18. Sometimes, it is convenient to combine both a declaration and definition into one statement, as shown in Line 22:

```
22          int age = 21; //alternative to line 14 + 18
```

3.19. If we want to print a message to the console, we can use the **`printf()`** function, as shown in Line 25:

```
25          printf("I will print my age!\n");
```

3.20. Notice how the format of the string inside the brackets is identical to the way you format it in Java. The "**`\n`**" is a newline character. This is the same as putting in a "Return/Enter" character on the console, so that the command-prompt doesn't appear immediately after the "**I will print my age!**" message.

3.21. We call a function in C in exactly the same way we call a method in Java, as shown in Line 28:

```
28          printmyAge(age);
```

3.22. Notice how we don't assign a variable to this particular function. This is because it doesn't ***return*** a value to the **`main()`** function. Instead, it simply performs some task. We know this, because the earlier function prototype showed that it has a data-type of void as shown in Line 6:

```
6     void printMyAge(int age);
```

3.23. If a function returned a value, its function prototype would start with that data-type. We can see in this in the case of the **`main()`** function, where it is declared as returning an integer, as shown in Line 12:

```
12    int main() {
```

3.24. We can see at Line 29, that the value it is actually returning is zero:

```
29          return 0;
```

3.25. As stated previously, notice that the program concludes with a final curly bracket at Line 30 (just like in Java), to complement the earlier open curly bracket immediately after **`main()`**:

```
30    }
```

4. Beginning learning C Programming

4.1. Open a **konsole**, and run the following command to determine what version of the **`gcc`** compiler you have on your system.

```
$ gcc -v
```

4.2. if you get the following error, this means you don't have the compiler installed.

```
bash: gcc: command not found
```

4.3. In the case of the above error, you will need to install **`gcc`**, **`gcc-c++`** and **`make`** command as follows. If you **didn't** get an error, skip to Step 4.4.

1. Run the **Mageia Control Centre** by clicking on the blue cog on the panel:

2. Click on **Install & Remove Software**.
3. At the top left, you should see a drop-down menu which says "**Packages with GUI**". Click on this and select the word "**All**" in ***this*** drop-down menu (***not*** in the box to the right), so that all package types will be visible. You should then have two drop-down menus, both of which say **"All"**. Then, in the "Find" box, type **`gcc`** and hit Return.
4. Tick the box beside the most-recent version of **`gcc`**.
5. If you get a message that RPMdrake needs to be updated or other messages about additional packages, answer YES/OK.
6. Further down the listing select **`gcc-c++`.** Tick the box beside the most-recent version.
7. In the find box, type **`make`**, and hit Return.

8. Scroll down and tick the box beside **`make`**.
9. If you get a message that RPMdrake needs to be updated or other messages about additional packages, answer **YES/OK**.
10. Now click **Apply**, and answer **YES** to any questions about additional packages, proceeding with the installation, etc.
11. When the installation of packages has finished, click Quit.
12. Now return to Step 4.1. above.

4.4. On page 2 of your Reflective Journal, after the table of questions, write down the command you typed in Step 4.1. above, along with the version number of your compiler that you see in the output.

4.5. Access the **`gcc`** manual page on your system by typing:

```
$ man gcc
```

4.6. From the list of most-useful options listed in the SYNOPSIS section of the manual page, list and briefly describe **three** of these options in your Reflective Journal. To search for an option type **`/option`** (followed by a space); for example: to look for the "**`-c`**" option, type:

```
/-c
```

4.7. **Note**: it is important to include a space after the letter "**`c`**". Hit the "**`n`**" key to search forwards for the next occurrence of what you searched for, and the "**`N`**" key to search backwards. When you are finished looking at the manual page, hit "**`q`**" to quit.

4.8. If you have difficulty finding these options, ask your Lecturer for assistance.

4.9. On the command-line, change into your **`programs`** directory (which you created in Practical 02), and create a directory called **`practical3`**. **Hint**: use the **`mkdir`** command.

4.10. Then change into this new **`practical3`** directory. **Hint**: the change directory command is **`cd`**

4.11. Using what you have learned in Section "3. Anatomy of a C Programming", start by attempting to write a simple "Hello World!" C program. Here is some guidance:

1. Using Emacs, create a file in your **`practical3`** folder called **`hello.c`**
2. All we want your program to do is to print a simple "Hello World!" message to the console.
3. Borrow only the essential code statements from the C program listing in the previous section to create the simplest program possible.
4. If you're confused, please ask for help!

4.12. To compile a program **`hello.c`** and create an executable called **`sayhi`**, you would type either one of the following equivalent commands:

```
$ gcc -o sayhi hello.c   OR   $ gcc hello.c -o sayhi
```

4.13. **Note**: you can also compile from within the Emacs editor. Do this by typing the following from within Emacs. Note that the "**M-x**" below refers to holding down the Meta key and tapping the "**x**" key once. On a standard PC, Meta is the **ALT** key. On a Mac, it is usually the **Option** key:

`M-x compile`

Delete the **"`make -k`"** text, and type the line below:

`gcc -o sayhi hello.c`

4.14. The advantage of compiling from within Emacs is that this editor provides *hypertext* for any errors encountered, which you can click on to show you the line of code where the error may occur. This makes it easier to locate the errors than when compiling on the command line.

4.15. From now on, please compile your C code files from within Emacs.

4.16. Recall from Practical 2 that when executing a program, you should precede the program name by a "`./`". Remember the **"."** is the short way of typing the path to your **current working directory (CWD)**. For example, to run the program **`sayhi`**, type the following:

```
$ ./sayhi
```

4.17. This is clearly more convenient than having to type:

```
$ /home/username/programs/practical3/sayhi
```

4.18. If this worked successfully for you, congratulations! You have just written your first C program. If it didn't work, ask your Lecturer for assistance.

4.19. **In your Reflective Journal**, on page 2, after the table, paste the text of your new **`hello.c`** program.

4.20. Now, open the C Programming tutorial located at the following URL, and then return to this handout to read the instructions:

https://www.tutorialspoint.com/cprogramming/index.htm

4.21. It is important to read through the ***descriptions*** in each section in the ***Tutorialspoint*** tutorial to gain a good understanding of the C programming language. This will ensure you don't end up with gaps in your knowledge. So, please avoid the temptation simply to jump to the code sections. The detail ***is*** important. Bearing that in mind, the first three sections provide a nice background on C. They are:

C – Home

C – Overview

C – Environment Setup

4.22. As you work through various sections in the tutorial, I want you to try out the programs by creating ***separate*** C files saving these in your **`practical3`** directory for each program listing. Try to choose file names that indicate a meaning for the program. Use Emacs as your editor, and compile each C file from within Emacs using the instructions above. DO NOT install the editor suggested by the ***Tutorialspoint*** tutorial!

4.23. The programming practice starts in the fourth section:

C – Program Structure

4.24. Once you complete all sections up to and including the section called "**`C – Functions`**", attempt the exercises below.

5. C Programming Exercises

5.1. In this section, you are going to start by writing some pseudo-code. Pseudo-code is a sequence of English-like short bullet points describing what you want your program to do.

5.2. **In your Reflective Journal**, on page 2, after the table, plan and write out the **pseudo-code** for a program:

1. that contains a loop which runs **20** times, and which prints to the console (screen) the iteration of the loop each time along with a newline character.
2. After the loop, the program should print out a "**`Goodbye`**" message and then terminate.
3. Consider whether it would be best to use a **for loop** or a **while loop**.
4. When you have finished planning your code in your Reflective Journal, and only then, write your program in C, using your pseudo-code as an aid. Call your program **`loopprinter1.c`**, and store it in the **`practical3`** directory.
5. Paste your code into your Reflective Journal, immediately below your pseudo-code.
6. Compile your program from within Emacs, and call the executable **`loopprinter1`**
7. Your program should print to the console something like the following:

   This is iteration 1

   This is iteration 2

   This is iteration 3

   …

   …

   This is iteration 20

   Goodbye

5.3. **In your Reflective Journal**, on page 2, after the table, copy and modify your pseudo-code from the previous exercise:

1. To include a call to a function, which performs the loop with the printing of the iteration to the console.
2. The main program should still print "**`Goodbye`**".
3. When you have finished planning your code in your Reflective Journal, and only then, write your program in C, using your pseudo-code as an aid. Call your program **`loopprinter2.c`**, and store it in the **`practical3`** directory.
4. Paste your code into your Reflective Journal, immediately below your pseudo-code.
5. Compile your program from within Emacs, and call the executable **`loopprinter2`**

5.4. If you get this far, return to the ***Tutorialspoint*** tutorial, and continue reading up to and including the section called "**`C – Arrays`**". Then, attempt the exercises below

5.5. **In your Reflective Journal**, plan and write the pseudo-code for a program, which:

1. *Declares* an array of integers of size **10**
2. Uses a loop to *define* each element of the array by giving it a definite integer value based on the index of the for loop; for eg: your program could simply define **`array[0] = 0`**, **`array[1] = 1`**, etc, so that your rule would be **`array[i] = i`**. Or, you could be more sophisticated and say **`array[i] = i*i`**.
3. Prints each value of the array to the console followed by a newline character.
4. When you have finished planning your code in your Reflective Journal, and only then, write your program in C, using your pseudo-code as an aid. Call your program **`mysimplearray.c`**, and store it in the **`practical3`** directory.
5. Paste your code into your Reflective Journal, immediately below your pseudo-code.
6. Compile your program from within Emacs, and call the executable **`mysimplearray`**

5.6. **In your Reflective Journal**, copy and modify your pseudo-code from the previous exercise, so that your new program:

1. *Also* adds up the values of each of the **10** integers (i.e. creates a **sum** of them) and
2. *Also* prints this **`sum`** to the console.
3. Note that in C, when one declares a variable, it does not automatically have a value equal to zero, unlike in Java. C has some quirks! So, if you're going to use a "sum" variable, it's important to give it an initial value of zero!!!
4. When you have finished planning your code in your Reflective Journal, and only then, write your program in C, using your pseudo-code as an aid. Call your program calcSum**`.c`**, and store it in the **`practical3`** directory.
5. Paste your code into your Reflective Journal, immediately below your pseudo-code.
6. Compile your program from within Emacs, and call the executable **`calcSum`**

5.7. Now complete the questions listed in the Table in your Reflective Journal and upload it to BrightSpace.