ShockSoc C Programming Support Sessions

Oliver Dixon
ShockSoc Technical Officer
od641@york.ac.uk

P/T/401 2pm–5pm
Spring Term, 2021–2022

Lab Script 01: Configuring the Environment and Understanding the Build Process

LaTeX document built January 31, 2022 on Linux.

Table of Contents

1 Introduction & Motivation

Welcome to the first Programming Support Session! Assuming the COVID-19 situation remains reasonably stable (i.e., we are not issued with another stay-at-home order), these sessions will be taking place every Wednesday afternoon in P/T/401, during the standard additional lab time. During this first script, you will be establishing a half-decent programming environment on a lab computer[1], understanding the build process of a simple C program, and finally writing and executing some short programs.

As the lab sessions progress into the Spring term, you are encouraged to cherish the time in the labs, undertaking one of three tasks (in descending order of importance).

- Work on your own projects;
- Progress through these lab scripts;
- Complete work from the *Introduction to Programming* module.

All PDF lab scripts, along with supplementary material such as code samples, is released under the public domain and published on-line at `https://www-users.york.ac.uk/~od641/pss/`, for the convenience of those preferring to work in their own time (as programmers often do)[2]. Should you have any remarks concerning these lab scripts, especially suggestions, corrections, or

[1] Almost all departmental computers, such as those on the first floor, will have exactly the same installation; these instructions are not limited to the fourth-floor labs.

[2] For the purposes of avoiding unnecessary confusion amongst first-year undergraduates, the raw LaTeX source code (.tex) files are not normally placed on the public HTTP instance. However, these can be quickly obtained by e-mailing Oliver Dixon for those with a special interest.

improvements, you are very welcome to contact Oliver Dixon using the e-mail address on the title page.

Please be aware that you are not expected to understand everything in these lab scripts. Some of the remarks, especially those contained in footnotes, are intended for readers already proficient in Linux and familiar with operating system internals to a reasonable degree. You may notice that footnotes are used very heavily, especially in the first few scripts. This is intentional and necessary, since this is a very large topic which may overwhelm less-competent readers. However, as we do not want advanced readers to be dissatisfied, 'interesting asides' are placed in footnotes.

1.1 A Criminally Abbreviated History of UNIX and C

In order to understand the development of C, you must first understand the development of the UNIX operating system. In the mid-1960s, Dennis Ritchie (`dmr`) and Ken Thompson (`ken`) were working alongside many computer scientists, electronic engineers, and mathematicians employed by Bell Telephone Laboratories in New Jersey to develop a revolutionary operating system: Multics. Whilst Multics was the first OS to implement many of the concepts still used today, such as a time-sharing process and memory scheduler alongside hugely increased data access abstraction, the implementation was large and messy, and also written for a bespoke GE-645 mainframe. After departing from the Multics team at Bell Labs, Ritchie and Thompson started work on a new operating system[3] that was small and clean, but retained the novel concepts introduced in Multics. Instead of being designed for a bespoke mainframe, Ritchie and Thompson decided to target the PDP-7 computer (and later PDP-11), as it was a machine rife with powerful hardware features, such as byte-addressability; see Figure 1 for a famous photograph of our two protagonists.

Although UNIX itself was written in assembly language for the PDP machines, Ritchie and Thompson needed a higher-level language in which simple UNIX utilities could be written. Thompson experimented with various techniques, and eventually settled on a simplified version of an informally specified systems programming language called *BCPL*, Thompson's simplification of which was called *B*. In the early 1970s, Ritchie became steadily dissatisfied with *B*'s performance problems and lack of full compatibility with the PDP-11 hardware architecture, and begun the development of a slightly improved version: `'B'++`, if you will. Fittingly, with the help of Brian Kernighan, another Bell Labs engineer, the name of *C* was chosen. By 1973, C was adequately robust to warrant the reimplementation of many core UNIX operations in the language, abstracting away from the dreadful PDP assembly language. (The various ad-

[3] The name "UNIX" was originally coined from *Uniplexed Information and Computing Service*, for which *Unics* is an acronym. This is a pun on Multics, which stood for the *Multiplexed Information and Computer Services*. Such technologically inspired comedic genius is second only to the *Generalised Information Retrieval Language System*, introduced by Dick Pick around the same time as UNIX.

[4] Image taken under CC-BY-SA-2.0 from Magnus Manske on Wikimedia Commons.

Figure 1: Ken Thompson and Dennis Ritchie—two of many original UNIX and C architects—working via teletypewriter (Teletype ASR 33) on a PDP-11 at Bell Labs in New Jersey, U.S.A.[4]

vantages and shortcomings of platform-independence shall be discussed shortly.)

At this point, it is worth noting that Kernighan and Ritchie published a book under Prentice Hall Software Series entitled *The C Programming Language* [3]. Colloquially referred to as 'K&R', this book is the greatest programming text ever released. In fact, it may be the greatest book ever released in any technical discipline, second only to Russell & Whitehead's *Principia Mathematica* or Donald Knuth's *The Art of Computer Programming* in an unfortunate joint first[5]. If you are a student looking forward to doing further reading, K&R is the only textbook you will need: it is rife with increasingly complex explanations and examples, and also written for programmers with no experience of C (although low-level programming experience is always helpful). The ISBN of the **Second Edition** is 978-0-131103627, and is available from any reasonable bookseller for around fifty pounds[6]. Any technical/university library will also stock copies available for borrowing, and, as an absolute last resort, free PDF scans of varying quality are easily found on the net.

[5]Electronic engineers true to their craft are likely to argue in favour of Horowitz & Hill over Russell & Whitehead. Thankfully, this course focuses on correctness.

[6]Make sure to avoid the first edition of this text, as it describes a version of C that is significantly different to the one in modern use. Specifically, it was before any sort of external standardisation (such as ANSI) had taken place, and was purely from the designs of Bell Labs engineers. Whilst very good, 'K&R C', as it is now termed, is no longer used in any aspect, aside from in obfuscated C competitions and the occasional round of code golf.

1.2 A Far Cry from Python: Why Use C at All?

Until now, many students have been exposed to only a tiny selection of very high-level languages; the typical list includes Python, Java, C#, Visual Basic, and perhaps even JavaScript. With the entertaining exception of Microsoft's VB.NET, all of these languages have their place, however none of them do anything to serve the needs of an embedded software engineer, as Electronic Engineering students tend to be. When writing software for embedded or performance-critical applications, layers of abstraction begin to present as more of a hindrance than anything else, as they tend to consist of three major issues:

1. With *very* rare exceptions, the above list of typically taught high-level languages all run under a hosted environment, as opposed to natively. This means that instead of being compiled into the native machine code of a particular processor (for example, ARM or x86), code is converted to an easier-to-parse intermediary form and executed under a virtual machine, which itself runs natively. In many ways, the 'intermediary form' (called *bytecode* in many communities, such as the Java Virtual Machine, or JVM) can be seen as an assembly language for the virtual machine[7]. In any case, for embedded applications, virtual machines tend to have far too much performance overhead and too large of a memory footprint, rendering them a completely unneeded layer of abstraction. (There are a plethora of advantages to execution under hosted environments, such as the existence of sandboxing and massively increased symbol-introspection capabilities for dynamic analysis and *in situ* debugging, however the overhead is still far too significant when working with system-on-chip boards with memory measured in the kilobytes. We will visit hosted execution again, but only for the purposes of debugging memory errors with *Valgrind*.)

2. As above, execution under a virtual machine tends to drastically reduce execution speed and increase power consumption. For example, research popularised by *El Reg* shows that when compared to C, Python is over seventy-five times less energy-efficient in performing a simple binary search, takes almost seventy-two times as long, and uses 2.4 times the amount of memory [4]. For embedded applications, this is simply unacceptable.

3. Platform-independence is not *always* desirable. By definition, platform-independence abstracts away from architecture-specific elements of a programming language, delegating them to various other intermediaries, such as vendor libraries implemented in assembly language with bindings to a target language, with the obvious examples being the AVX and SSE processor extensions. While increased portability tends to improve the lives of programmers and users alike, in the context of embedded systems, software and hardware work very intimately: far more than anything seen

[7] Hopefully it goes unsaid that this introduction is very crude and criminally oversimplified, as was the historical overview of UNIX. There are many distinctions between an "interpreter" and a virtual machine, however they are beyond the scope of the material at this stage.

in typical user-space applications. As such, this layer of abstraction can prove unhealthy, and may render some common heavy optimisations impossible, such as writing to individual registers or working with processor jump tables. Some operations, such as disabling interrupts or switching to supervisor mode, are seldom possible outside of a processor's assembly language[8].

While far from perfect, the use of lower-level languages eliminates many of these issues by using a compiler, linker, assembler, and various other utilities, to convert and combine human-readable source code into executable native machine code for a specified architecture. We will now discuss the build process of a typical program in slightly more detail.

2 A Simplified Build Process: Step-by-Step

The build process typically consists of two major stages: compilation and linking. Respectively, these discrete stages can be abstractly defined as "converting the source files to machine code", and then "combining all the machine code together to produce a single binary, resolving cross-references and aligning data". (There are a few different types of linking in common use, however we will only consider *static linking* in the text body[9].) In any case, the general purpose of a *build* is to take one or more source files and combine them into a single executable; this resultant binary may or may not have a well-defined entry point[10]. This is a long and arduous task with many considerations to be made, a slim minority of which will be covered here. Formal compiler theory is a very active

[8]Do not worry if these particular examples don't make sense yet; their meanings will become clear in time.

[9]In this introductory document, static linking is assumed for reasons of simplicity. In reality, static linking is rarely used for common user-space applications, since resultant binaries become extremely large for the most simple of programs. Dynamic linking and shared linking are more common approaches, in which libraries are either loaded on-demand via an OS-provided API, or immediately linked upon execution of the client program. The latter approach is especially popular in UNIX-like systems, with *shared object* (`.so`) files typically residing in `/usr`. To find all of them, try running "`find /usr/lib{,64} -type f -name \\*.so 2>/dev/null`" in a shell. Interestingly, the linker binary is still termed `ld` on the vast majority of systems, due to its original name of "the *loader*".

[10]In the context of binaries, an "entry point" is the address of the function to be called upon execution of the program. Binaries need not have an entry point to be valid. In fact, most binaries installed on most computers do not have entry points, as they are libraries designed to be called from client code, and not to be executed as standalone programs. In the context of C, compilers normally use the `main` function as an entry point to be encoded into the `_start` routine. For truly embedded applications, simple microcontrollers tend to jump to address `0x00`, interpret the data as instructions, and execute. In that sort of situation, it is the responsibility of the programmer to write a linker script that ensures the desired entry function (of an arbitrary name) is located at `0x00` in the flashed binary. Furthermore, it may be helpful to realise the entirely decoupled nature of the compiler and the linker: almost every toolchain allows executing the compiler but not the linker, or the other way around. We explore this later when requesting the builder to output the ARM assembly language as opposed to a well-formatted ELF binary.

area of research and comprises a significant portion of Computer Science degree courses; a strong pure mathematical background is also required for a full understanding.

2.1 The Compiler

The compiler is typically a very large program, tasked with performing a few major subtasks: pre-processing, lexical analysis, syntax analysis, optimisation, and finally code generation. As steps are executed in-sequence, a basic description of each stage is an appropriate way to think about the role of a compiler:

- **Pre-processing.** "Preprocessor directives" are an integral part of any C program. They serve as simple, compile-time instructions to be evaluated at the first stage of the build, and are identified with a leading pound sign. Types of preprocessor directives are few in number, but are enormously useful for declaring named global constants (`#define`), including external source files inside an independent translation unit (`#include`), and also forming basic conditionals (`#if`, `#else`, and `#endif`). They also tend to be used to implement compiler-specific extensions, such as `#warning` to issue a build-time warning to the shell, usually as the result of a conditional. (The fatal analogue `#error` is standard, and will immediately halt a build.)
- **Lexical Analysis.** All programming languages are composed of 'tokens', where a 'token' is the most primitive unit of an expression. C has five types of tokens, all of which must be recognised by the lexer:

  1. Primitive keywords, such as `if`, `for`, `else`, and `while`;
  2. Identifiers, such as variable and function names;
  3. Operators, such as `+`, `-`, `&`, and `|`;
  4. Literals, such as 'magic numbers', or those defined with `#define` (after the pre-processing stage, these two are identical). These may also include strings encapsulated between a couple of speech marks;
  5. Special characters, such as parentheses of various forms.

  The lexer also identifies comments, which, unless instructed otherwise by the compiler, will be removed along with any unnecessary whitespace[11].
- **Syntax Analysis.** At this stage, an ordered array of tokens has been compiled, however the compiler has no notion of their validity. The purpose of the 'parser' is to perform syntax analysis on the tokens, ensuring

[11] It may be worth noting that non-standard keywords are allowed in the implementation namespace, or after the inclusion of a non-standard header. For example, after the inclusion of `8086.h`, `far` and `huge` are defined as non-standard keywords to describe a pointer. Various compilers may also define additional keywords, such as GCC's `long long`, however these should generally be avoided unless you have an absolute certainty of the build system's extensions.

they are all of correct form, and are organised in accordance with the specification of the language [2]. At this point, any syntax issues will be raised by the compiler. Once the source is known to be valid, it can be converted into an intermediary representation (IR), as mentioned earlier. At the time of writing, every moderately serious C compiler uses an abstract syntax tree (AST) to store the IR[12].

Use of an IR also allows for great versatility in toolchains, such that developers are spared the pain of writing new build systems for every language. In the case of GCC or Clang, adding support for a new language requires only the implementation of the tokeniser and the syntax analyser. Once a translation unit is in the IR, optimisations and code-generation can be unified.

- **Context-Sensitive Analysis.** Having the program in its IR also allows quick binary traversal of all execution paths, enabling the compiler to perform checks for redundant variables or obviously superfluous code, such as '`if (0) x = 0;`'. Depending on compiler flags, this may result in the silent removal of redundant expressions, or an explicit warning being reported. Potential trouble-sources, including the likes of declaring uninitialised pointers or obvious errors[13], are also reported at this stage, but are not typically presented as errors.

  Debugging information can also be packed into the binary at this stage, if requested by the compiler. As C does not support introspection, unobfuscated names and types can be placed into the *global symbol table*, which may be interpreted by a debugger to look-up identifiers and types. If a very high level of debugging information is requested, the original source code with line numbers, whitespace, and commenting intact may also be packed into the binary.

- **Optimisation**. Optimisation is an extremely heavily researched area of compiler theory, and for good reason. Optimising an AST is not a black-and-white situation, as have all the previous stages. A compiler, unless told otherwise, will typically choose a sensible balance between binary size and execution speed, often veering towards the latter. Common optimisations include loop-unrolling, the reduction of mathematical precision, and the omission of various safety features; all of these will be examined in detail throughout the coming sessions, as it is very difficult to understand these techniques before learning any C.

  For those interested in further reading, many impressive bit-level optimisations for very common operations (such as calculating $\log_2(2^n)$, $n \in \mathbb{Z}$) are

[12] Use of an AST is advantageous for various reasons, however the primary lure of a tree-like structure is shown in its ability to implicitly represent many details that would have to be expressed explicitly in source code, such as parenthetical structures. Similarly, many common constructions are easy to represent in the form of a tree, as typical statements such as *if-then-else* compounds may be represented as single nodes with multiple branches.

[13] ... such as dereferencing `NULL` or accessing an array with an unreasonable index.

available on the net [1]. Intelligent compilers will spot cases in which these optimisations may be useful, and substitute an appropriate compound. Iterative optimisation is also often performed, such that optimised compounds may have their loops unrolled, or branchless substitutions made.

- **Code Generation.** Finally, with a correct and optimised IR AST, native code with architecture-specific opcodes and syntax must be generated and packed into a binary. These binaries are known as 'object files', as they have yet to be linked into a single executable by the linker.

  A compiler will usually generate native code for its host architecture (e.g., compiling on x86 would create x86 binaries), however cross-compilation is also possible, such that a compiler generating 32-bit ARM binaries could be executed on a SPARC machine. In the context of embedded software engineering, in which development work is typically undertaken on an x86 machine with a non-x86 target, cross-compilation is a vital part of the toolchain[14].

2.2 The Linker

After compilation of the source files, the linker is executed with an array of relocatable object files. Akin to the compiler, the linker performs a number of major subtasks to combine the provided object files and libraries into a single program: symbol resolution, section alignment, and initialisation. Again, these are typically conducted in sequence.

- **Symbol Resolution.** As previously mentioned, out-of-scope symbols remain as foreign unknowns, of which the compiler is only aware by name. At this stage, the linker iteratively cross-references symbol names between object files and ensures they all have a unique address. Assuming all symbol references can be resolved and mapped to a unique address, the sections from the object files are now concatenated.

- **Section Alignment.** All concatenated sections must now be aligned and assigned a unique address, usually derived from a single base address[15]. (The concept of a 'section' will be explored in more detail shortly, once we start inspecting compiler behaviour by decompiling binaries and cross-referencing the C source.)

- **Initialisation.** Despite being one of the most important stages for linking binaries to be flashed to embedded systems, data section initialisation is far beyond the scope of an introductory document of this nature. For an elementary understanding, you just need to know that initialised data is

[14] Students familiar with Arduino boards will, likely unknowingly, have used a cross-compiler embedded into the Arduino IDE to compile for an ARM or AVR target on an x86-based host.

[15] Section base addresses may be plentiful, and do not usually float. Non-volatile memory typically stores base addresses for code and data sections, such as the program and the call stack.

usually moved to non-volatile flash memory, and that constant data (such as code) may be occasionally moved to RAM for increased fetch-execute performance.

Assuming nothing untoward occurs, the linker will output a binary in a target-independent common format. The usual list of contenders includes ELF (for UNIX-like systems), DWARF (for enhanced debugging), COFF (for legacy UNIX-like systems), or PE (for Microsoft Windows)[16].

2.3 Diving into the Build Stages: A Practical Example

NOTE: this section is intended for advanced readers interested in practical inspections and decompilations of simple programs. If this does not describe you, please skip this section, as there is a non-zero chance of you becoming irreconcilably confused for little benefit. A basic knowledge of C and ARM is also assumed, although not necessary.

While the primary function of any build system is to take a collection of source files (or 'translation units') and output a binary, virtually every compiler will contain options for outputting its intermediary stages. In addition to proving rather useful for debugging and performance-analysis, such options are invaluable teaching tools. In GCC and Clang, the `-S` option can be used to "only run the preprocessing and compilation steps"; the result being an assembly language (`.s`) file. For a quick example, consider the following C function to swap two words *in situ*: `*c1` and `*c2` (this algorithm should never be used in practice):

```
#include <stdint.h>
typedef uint32_t word;

/* Swap the words referenced by c1 and c2, assuming c1 != c2. */
void swap ( register word * c1, register word * c2 )
{
    *c1 ^= *c2;
    *c2 ^= *c1;
    *c1 ^= *c2;
}
```

Listing 1: A reasonably ridiculous swap function, valid for differing `c1` and `c2`.

Requesting a decompilation from an ARM compiler[17] results in the following

[16] ELF: Extensible Linking Format; DWARF: Debugging With Arbitrary Record Formats (debated); COFF: Common Object File Format; PE: Portable Executable. Before flashing these onto a microcontroller with a programmer, these target-independent binaries must be converted to a native format, which inconveniently differs between microcontroller vendors.

[17] This particular example was compiled for an ARM7TDMI 32-bit processor, which is a decent processor with a pleasant instruction set. This proves very useful for educational purposes; more information can be found on-line: `https://www.ecs.csun.edu/~smirzaei/docs/ece425/arm7tdmi_instruction_set_reference.pdf`. Further technical documentation can be found on the infinitely useful ARM Developer website: `https://developer.arm.com/documentation/dvi0027/b/arm7tdmi`. This particular assembly listing was produced with GCC, on a Gentoo Linux x86_64 host: "`arm-none-eabi-gcc -S -c -O3 swap.c`".

exciting listing (overzealous comments have been added for clarity):

```
swap:
    ldr r2, [r1]    ; Dereference a parameter into r2
    ldr r3, [r0]    ; Dereference the other parameter into r3
    eor r3, r3, r2 ; XOR the two dereferenced parameters into r3
    str r3, [r0]    ; Store the XOR into the original first param.
    ldr r2, [r1]    ; Superfluous. (A weakness of C pointers.)
    eor r3, r3, r2 ; XOR r2 and the result of the last XOR into r3
    str r3, [r1]    ; Store the XOR into the original second param.
    ldr r2, [r0]    ; Dereference r0 into r2.
    eor r3, r3, r2 ; XOR the results with each other
    str r3, [r0]    ; Store this final result into the first param.
    bx  lr          ; Return to the caller (the address of which is
                    ; stored in the link register)
```

Listing 2: An ARM assembly language representation of the swap function.

While we are not going to go deeper into this particular example, these short listings hopefully provide a taste of the potential power that can be derived from exploiting the internal representations of a build system[18]. What do you think the assembly listing would look like without the `register` keyword in the C source, or if we were to add a `c1 != c2` sanity check? What about if we were to replace the parameter datatypes with something not equal to the size of a native word, such as a `char`?

3 Configuring and Testing the Environment

By this stage, the studious of readers may have noticed a preference towards UNIX-like systems. Indeed, virtually all low-level systems and embedded programmers have a strong bias toward UNIX clones, coupled with a particular hatred for all things Microsoft. While Linux is a monolithic and politicised mess of an operating system[19], it is very ubiquitous and a decent clone of the original UNIX principles, which render low-level development very pleasurable (or relatively painless). A distribution of Linux (Debian) is also installed on all computers in the Department of Physics, hence its use in these lab sessions.

3.1 Booting Linux and Accessing the Terminal Emulator

In order to access Linux, restart the computer from Windows and use the arrow keys to select "Ubuntu Linux" (or some trivial variation thereof) from the GRUB bootloader menu, which can be identified by its trademark purple background. With the correct entry highlighted, hit the enter key to boot into Linux. After

[18] ... for which you should always RTFM.

[19] For those infatuated with UNIX principles but tired of Linux's nonsense, perhaps you may find love with an open-source version of BSD, or even a copy of Andy Tanenbaum's MINIX 3? The latter is a particularly interesting clone of UNIX with a microkernel design, designed to be small, clean, and secure by running programs in the highest possible protection ring. If you are a member of the *chosen few*, installing a copy of Bell Labs' Plan9 inside a virtual machine may prove amusing; it is occasionally well-described as "UNIX on [anabolic-androgenic] steroids".

a few seconds, you should be able to log in with your standard University ID and password (for example, `od641` or `mkw525`)[20]. Once you have navigated to the desktop, it is a good idea to familiarise yourself with the terminal emulator (named as such due to not being a physical serial port connected to a teletypewriter[21]); most of the work during these labs will be conducted via a textual command-line interface, as opposed to a GUI. To launch the standard GNOME terminal emulator, depress the `Control`, `Alt`, and `T` keys simultaneously. (An alternative and slightly more 'authentic' solution is the use of the `/dev/tty`*N* devices, which can be accessed with `Control`, `Alt`, and `F<1-6>`, with `F7` used to return to X11. However, if you wish to use any graphical applications, such as a web browser or GUI text editor, this solution becomes fairly impractical rather quickly. If you are in favour of using the `/dev/tty`*N* devices, you may find the `tmux` or `screen` programs useful for terminal pseudo-multiplexing.)

Once you have a terminal emulator at your disposal, ensure the core development programs are available in the `$PATH` by executing the following command: `whereis -b gcc gdb valgrind`. If any lines draw a blank (such as "`gcc:` "), please inform the PSS author, as you will be incapable of doing development work on the machine.

3.2 Selecting a Text Editor

One's choice of text editor has long proven to be a contentious point in technical circles, with all sides of the debate insisting the validity of their preference; or, more amusingly, the ridiculousness of everyone else's. In particular, programmers tend to be divided between Bill Joy's `vi` (or more precisely, Bram Moolenaar's `vim`), and GNU's `emacs`[22].

Unless you already have a preferred editor, you are recommended to use a simple graphical editor with good syntax-highlighting support out-of-the-box. For Ubuntu/GNOME systems, this is `gedit`. (If you are sensible, you will grow to hate `gedit` with the passion of one-thousand O-type stars, however it is an adequate starting point for those new to development on Linux.) To execute `gedit`, enter `gedit &` at a shell (with the ampersand used to detach the process from the shell), or use the GNOME Launcher. From there, the operation of the editor is reasonably self-explanatory. For the first stage, you must ensure that your compiler is functional and capable of locating the standard libraries; this can be achieved with a simple "Hello, World!" test program.

[20] We have occasionally experienced issues with logging into the Linux machines in the first-floor Physics PC labs. If this is the case, please contact IT Services, as there is little anyone can do, aside from forwarding your complaint to IT.

[21] The PSS author has a Tekade FS 200Z teletypewriter capable of connecting via USB to a Linux machine on the V.10 interface; he would be pleased to discuss it with anyone interested. The 1970s-era dot-matrix TTY is currently situated in a Halifax College first-floor bedroom and weighs almost 70kg!

[22] This feud has given rise to the so-called 'Editor War', which has a rather entertaining Wikipedia article by the same name. Campaigning for the superiority of `emacs`, GNU founder Richard Stallman has founded the 'Church of Emacs', over which he presides. Conversely, Bram Moolenaar has declared himself the 'Benevolent Dictator for Life' of matters concerning `vim`.

3.3 Writing a "Hello, World!" Test Program

```c
#include <stdio.h>

int main ( int argc, char ** argv )
{
    puts ( "Hello, World!" );

    for ( int i = 1; i < argc; i++ )
            puts ( argv [ i ] );

    return 0;
}
```

Listing 3: A typical "Hello, World!" test program, printing any command-line arguments in sequence.

To build this program[23], save it in a sensible location on your University filesystem, such as `~/pss/helloworld.c` (where `~/` is your home directory), and navigate to the same location in a terminal using the `cd` *<path>* command; to view the contents of a directory in a pleasant format with human-readable file sizes (using SI multipliers as opposed to a number of bytes), use `ls -alh`[24]. To build this program, execute "`gcc helloworld.c`" in the same directory as the source, which is assumed to be named `helloworld.c`. If GCC returns any output, make sure you have copied everything correctly; if compilation still fails, please contact the PSS author, as there is likely an issue with your machine.

To run the program, enter "`./a.out`". You should get a pleasing message: "`Hello, World!`". To ensure command-line arguments are received correctly from the shell environment, try executing "`./a.out PSS Test`". If all goes well, each command-line argument—delimited by spaces for every entry after the executable name—should be printed on a new line after the initial message. If you still have time, try the following short exercises:

- Change the "Hello, World!" message to something slightly more exciting and cultured, such as "G'day mate!".
- Try to output a multi-lined fixed message. You may want to read about 'ANSI escape sequences', and how they are interpreted in C strings.
- **(Difficult.)** Can you find a way to print the list of command-line arguments in reverse order? [Hint: `argc` stores the number of arguments.]

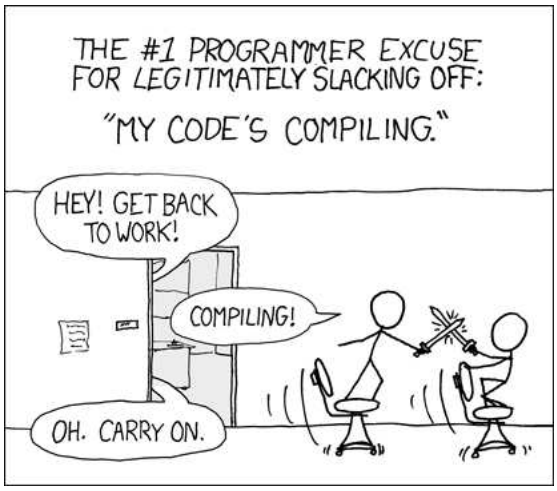
[23] You are not expected to understand this program; it is merely used to test ensure that the compiler is working to some acceptable degree. Although you are encouraged to input the program manually, to get a *feel* for writing C, you may also copy-and-paste `helloworld.c` from the PSS lab scripts website.

[24] It may prove useful to 'alias' this to a shorter command, since you will be typing it relatively often. To do this for the current session, enter "`alias ll='ls -alh'`". If you would prefer this persist throughout every Bash session, append it to your `~/.bashrc` file with "`echo "alias ll='ls -alh'" >>> ~/.bashrc`". Be careful to use a triple greater-than sign here; using only two of the symbol will irreparably overwrite the entire file, and it is rather important!

3.4 Next Week...

During next session, we will start writing some C. We will be examining the meaning of a *data type*, the concept of *scope*, correct conventions regarding *commenting*, and also introducing the idea of a *function* to write modular code.

See you then, I hope you enjoyed the first PSS script.
PSS Author, Oliver Dixon



XKCD #303: "Compiling". `https://xkcd.com/303/`

I would also like to extend my gratitude to Adam Gottesmann, Tom Mason, and Will Hinton, for reading drafts while providing invaluable and continuous feedback.

Referenced Works

[1] Sean Eron Anderson. *Bit-Twiddling Hacks*. 2005. URL: `https://graphics.stanford.edu/~seander/bithacks.html`.

[2] ISO/IEC JTC 1/SC 22. *The C18 Standard*. ISO 9899. June 2018. URL: `https://www.iso.org/standard/74528.html`.

[3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd ed. Englewood Cliffs, N.J.: Prentice Hall, 1988. ISBN: 978-0-131103627.

[4] Rui Pereira et al. "Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?" In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267. ISBN: 978-1-450355254. DOI: `10.1145/3136014.3136031`.