MAP55613 - C Programming

M.Sc. in High Performance Computing
School of Mathematics

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Richie Morrin - rmorrin@maths.tcd.ie

Michaelmas Term 2023

Tentative Outline

- L1: Introduction
- L2: Variables, scope, operators
- L3: Type conversions, Control flow.
- L4: Functions.
- L5: Recursion, Pointers & addresses
- L6: Arrays, memory layout
- L7: Memory allocation
- L8: Some sorting algorithms (Maybe moved to later)
- L9: I/O
- L10: C-strings, Timing, RNG, GSL
- L11: structs.
- L12: Compilation stages. Multiple files.
- L13: static, extern, command line args
- L14: Multi-dimensional arrarys, linked lists.
- L15: Doubly linked list, review pointers & memory
- L16: Optimisation
- L17: MKL, BLAS
- L18: LAPACK
- L19: Function pointers, variadic functions
- L20: System calls, fork, pipe
- L21: Named pipes, semaphores, shared memory
- L22: (pthreads?) Quick review

TBC: Likely have 6-8 assignments in total.
There will be an exam this year

Important notes for this course.

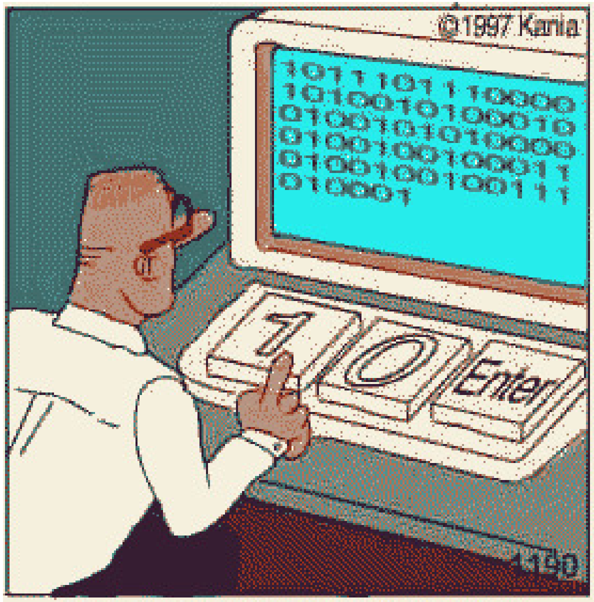
- Generally, students score very high in this course
  - There have been some that failed it!
  - ⇒ Use the help rooms/tutorials.
- For Assignments:
  - Don't blindly copy from other students/internet/AI tool.
  - Make sure that what you hand in compiles even if it does not run correctly. (Ideally, test on `chuck` before submission)
  - Don't include object files or executables
- Other tips:
  - Get used to the TCHPC environment as soon as you can.
  - Learn an editor.
  - Use the help rooms/tutorials. (Included twice on purpose!)

By the end of the module you should be able to do well on an interview for an entry level C-programming job.

This Lecture

After this lecture you should:

- Understand some of the basic differences between lower-level and higher-level languages
- Know a brief overview of history of C
- Understand why we might choose C
- Have an idea of what a compiler does
- Be able to write, compile, and run your first C program



Real programmers code in binary.

Languages

Programming languages span a spectrum from low-level languages that have simple "vocabulary" - and are "closer" to the hardware - all the way to feature-rich high-level languages - which are closer to spoken language.
In general:

- Lower-level languages
  - Less language features to learn.
  - You have more direct control over what is going on in the hardware.
  - But you need more of an understanding of what is happening.
  - Faster.
- Higher-level languages
  - Can be easier to follow and understand larger programs.
  - Takes care of some of the work automatically for you.
  - Easier to write, reuse, and maintain your code.
  - More layers of abstraction before you get to the hardware $\rightarrow$ so can be slower.
  - Can be more portable.

Languages

Assembly (Just for display purposes)

```
        .file   "HPC.c"
        .section        .rodata
.LC0:
        .string "Hello World"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-39)"
        .section        .note.GNU-stack,"",@progbits
```

Languages
Perl (For Comparison)

```
print "Hello World"
```

Translating to machine code

How do we get from higher-level code down to code the hardware understands?

- Traditionally[1] - two general approaches
  - Interpreted - at "run time"
    - Faster to write and test. (Useful for prototyping)
  - Compiled - in advance.
    - Compiled code runs faster.
    - Can share executable rather than source code.

[1] I say "traditionally" as there are some examples where a mix is used
Note also, technically the language itself is usually not specified so that it must be either interpreted or compiled . . . these are just ways to translate the language down to machine code

Why choose C?

- Legacy. A lot of HPC/scientific/technical software is written in C
    - Unix (re)written in C. Linux kernel written in C.
    - Legacy also means lots of existing libraries.
- C is a compiled language.
- Relatively "low-level". Closer to the hardware.
- Simple (procedural) and efficient so code is often faster.
- Syntax is foundation of many newer languages[3] (C++, Java, ...,).
- Popular language - lots of tutorials and websites online.

[3] N.B. C-shell (csh) is not the same as C. It was called that because it used similar syntax. However it is a very bad shell to try to program in. Avoid if you can!

History of C

Early 1970's	Created by Denis Ritchie in AT&T Bell Labs
1978	K&R C published
1989 (90)	C89 standard (C90). Often refered to as ANSI C.
1999	C99 standard
2011	C11 standard
2018	(C17) C18 standard
~~2023~~ 2024	C23 standard

Note about standards

- Basically a set of rules that define how language should work.
- Compilers should conform to standards. However they can also have extensions or non-standard features - can cause issues with portability. We will aim to use standards compliant code, and best practice, in the course.
- New features are added and older practices may become obsolete or considered bad practice.
- C is backwards compatible but *you might see code that looks slightly different*, especially in older books.
- Not everything is defined absolutely in standards - some details are implementation dependent - but you don't need to worry too much about that for now.

Note about compilation

General idea is to let the compiler do the hard work whenever possible

- Compiler job has two "levels"
  - Translate your high level code down to instructions the hardware can follow
  - Figure out what you meant to do and try to do it efficiently. Ignore this for now
- More later: When we refer to "compiling" we generally mean a 4-stage process - Pre-processing → compilation → assembling → linking.
- When we compile we can sometimes get warnings, or errors, or both. Compilation will still complete in the case of only warnings - however do not ignore them! Run-time errors are much more difficult to solve. Let the compiler do the work!
- You can also tell the compiler to be more pedantic with its warnings. Use these flags! E.g `-Wall -Wextra`
- Makefiles - you will learn about these in another course. They will simplify your life for larger projects.

Learning C

Q: What is the best way to learn how to program in C

A: Write lots and lots of C programs. Make mistakes, find bugs and fix them.

A useful reference is `https://en.cppreference.com/w/c`

Note that one good, and often forgotten, resource for learning about different C functions is your standard Linux man pages (section 3). To access these, you will type `man -s 3 functionname` where *functionname* is the name of the function that you want.

Connecting to TCHPC

For **today**, I assume zero experience and that you are running Windows!

- If you have *nix or macOS, you probably already have a compiler installed. You can edit and save on your own machine for today
- However, make sure you get access to the TCHPC system.
- In general, your code/assignments will need to be able to run on `chuck` so that I can check them

You will need to be able to connect to chuck.

- Need an SSH client - e.g. putty `https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html`
- If outside TCD firewall, you will need to first connect to `rsync.tchpc.tcd.ie` (or `maths.tcd.ie`) and then jump to `chuck` (firewall reasons)

Editor

- You can choose whatever editor/IDE you would like.
- Historically, “editor war” between `vi` and `emacs`
- My personal choice is `vim`. Steep initial learning curve. You can type `vimtutor` to learn basics
- For purpose of class today I will use `nano` as it should be usable by those who don’t know any other editor

Hello World

Open your preferred text editor (or try to follow what I am doing with nano Type the following and save it as hw.c

hw.c

```c
#include <stdio.h>

int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

Now to compile

1. Open a terminal
2. cd to the directory where hw.c is saved.
3. Type ls to list the contents of your directory.
4. Now type gcc hw.c.
5. If you get errors (or warnings) fix them!
6. Type ls again. You should see a new file called a.out. This is the default output name for gcc.
7. Type ./a.out to run your program.
8. Re-run (4) but this time choose the output file name, e.g. hw.
    - Type gcc -o hw hw.c.
9. Type ls again. You should see another new file called hw.
10. Type ./hw to run your program.

Anatomy of Hello World

hw.c

```c
/* Useless Comment Example */
#include <stdio.h>

int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

- Anything between /* */ is ignored as a comment.
- The second line includes necessary information from a file stdio.h. More later.
- main is a special name reserved for the function where execution starts.
- int denotes that main returns an integer.
- Code blocks contained within {}. These define a *scope*.
- Instructions are separated by semi-colons (;).
- printf is a library function which prints output to the standard output i.e. terminal. \n prints a newline character.
- Whitespace does not matter to the compiler - However it will matter to the person reading your code!!!

General Tips

- Don't get frustrated if your code does not compile straight away and if you cannot find a bug. It comes with practice.
- Understand the importance of commenting code. If you want to get fancy, you can learn how to use utilities such as Doxygen for creating nice documentation.
- Choose a coding style and be consistent.
- Don't ignore compiler warnings when compiling.
- As mentioned previously: learn how to use Makefiles when they are introduced later.
- Learn how to use Source Code Control (e.g. Git). You will also cover this in other courses.
- Learn how to use features of your editor!

Recap of L1

- Introduced concept of high-level and low-level languages
- Introduced the concept of compilation
- First program.

Try to make sure before Friday's class that you:

- can connect to TCHPC system
- can compile and run "Hello World"

Let me know at the start of that class if you have any issues.

Over the next week or so:

- Start learning an editor if you don't already have a preference

If you want a quick overview of compilers, you could take a look at e.g.
`https://www.youtube.com/watch?v=0A079QpDAzY`