C Programming Tools: Part 1

Building and Using your own Toolkit

Duncan C. White
d.white@imperial.ac.uk

Dept of Computing,
Imperial College London

29th May 2018

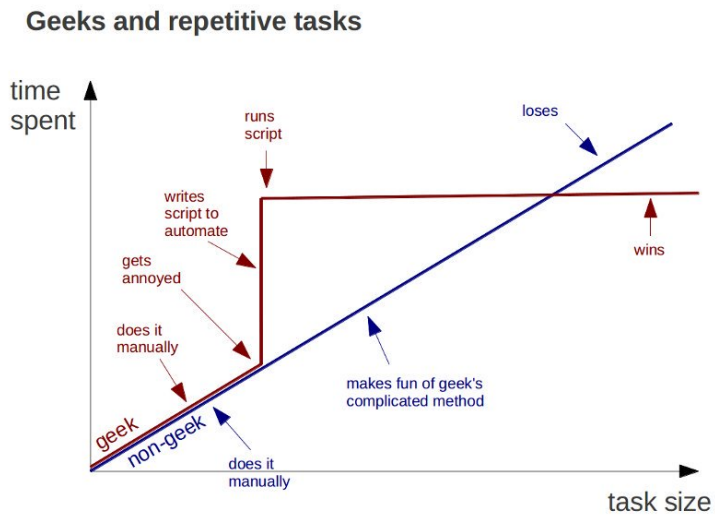
- As programmers, you will learn many languages over your career. Right now, you're learning C.
- Hopefully you'll learn something from each language. But some languages flower briefly and then die, others endure. C is likely to endure. It's endured for over 40 years already!
- When learning a new language like C, there are several steps to achieve basic competence:
  - Learn syntax.
  - Learn semantics.
  - Learn the tricky bits eg. pointers.
  - Learn the standard library.
  - Learn how to write multi-module programs.
  - Learn the idioms and best practices.
  - Learn to avoid the traps and pitfalls.
  - Learn how to write portable code.

These 4 lectures, today and the next three Thursdays, answer:
What comes after basic C competence?

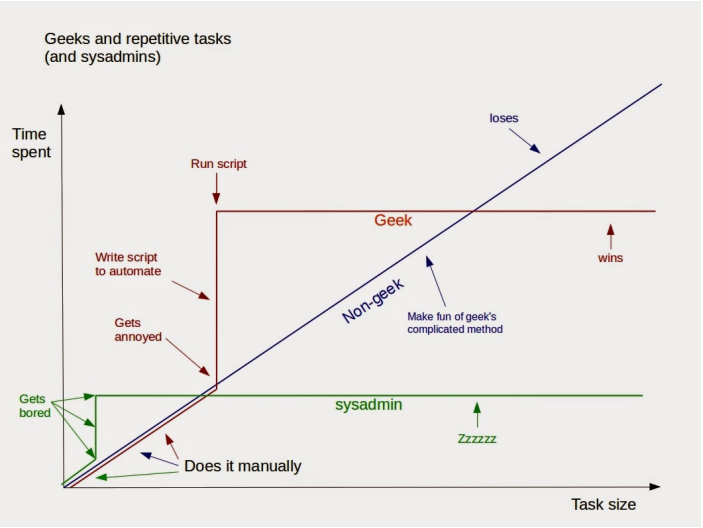
- Craftsmanship!
- Build your own toolkit of:
  - Useful tools,
  - Useful libraries,
  - Craft skills.
- To make C programming easier and more productive.
- When necessary: build your own tools!
- Core Principle: Ruthless Automation.
- Doing something boring and repetitive? Can I save time by automating this?

Or, to put that another way:
(As seen on the walkway last year).

Or, to put that another way (thanks due to SwissMiss):



Or, adding SysAdmins into the mix:



Today, we'll cover:

- Programmer's Editors: Use a single editor well.
- Automating Compilation: Use make.
- Multi-Directory Programs and Libraries: How to lay out programs in multiple directories, a Makefile per directory.
- Automating Compilation: An alternative tool called CMake.

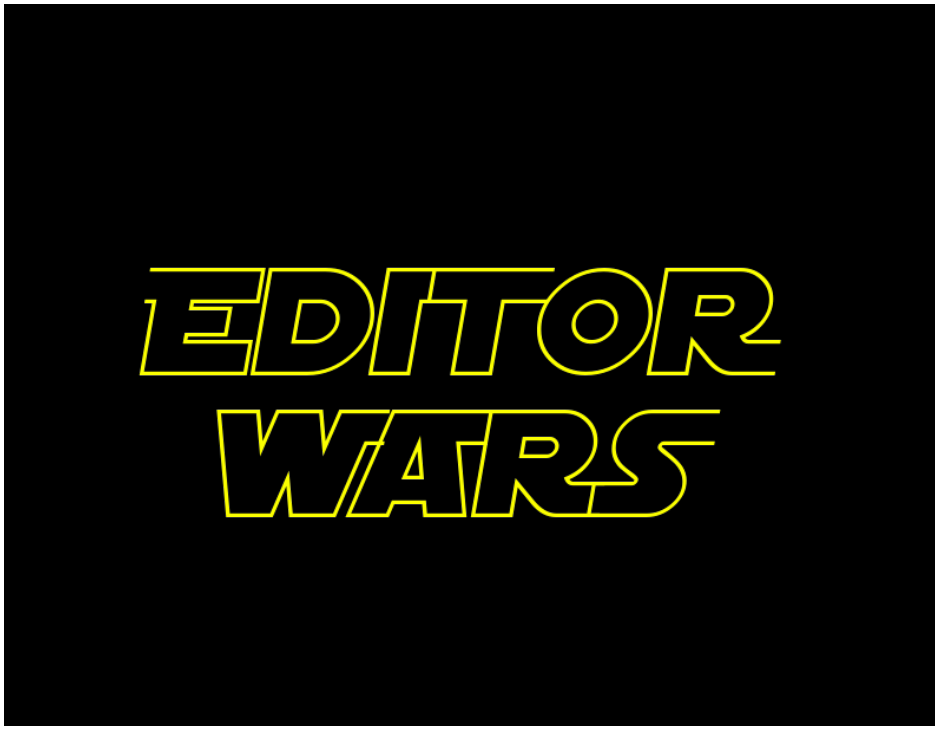
Notes:

- I strongly recommend The Pragmatic Programmer (PP) book, by Hunt & Thomas. The woodworking metaphor - and a series of excellent Programming Tips - comes from there.
- I also recommend The Practice of Programming (PoP) book, by Kernighan & Pike. Both books are brilliant expositions of expert-level programming craft.
- There's a tarball of examples associated with each lecture, as a shorthand tarball 01.intlist refers to the directory called **01.intlist** inside the tarball. Each directory contains a README file.

- Hunt & Thomas write (in Tip 22):

  Use a Single Editor Well: The editor should be an extension of your hand; make sure your editor is configurable, extensible and programmable.

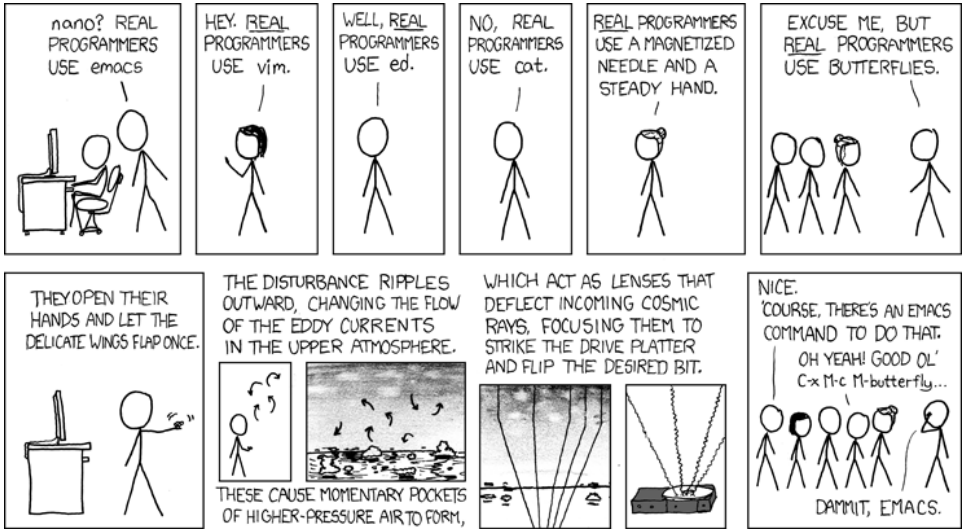
- As a programmer, you will spend years of your life editing programs.
- Coding might be 80% thinking and 20% typing, but your typing must not interfere with your thought process.
- So: Explore a few editors, choose one, and spend time becoming expert in it.
- That includes: learning how to plug external tools in.
- It's more than my life's worth to tell you which editor to use.
- Why? Because programmers are notoriously sectarian when it comes to..



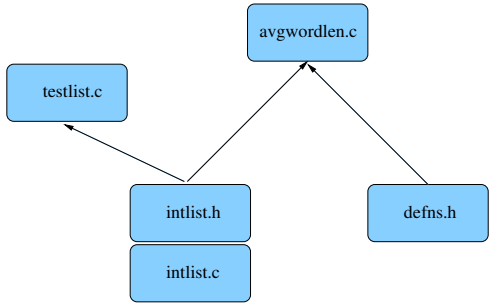
- The leading Programmer's editors are (probably) vim and emacs:

- IDEs such as Eclipse and CLion provide an editor, an automated compilation system and a debugging environment. If you're going to use an IDE, learn how to use it well, and how to extend and program it.
- Note that Hunt & Thomas aren't much in favour of IDEs. Neither am I:-)

Actually, it's well known that Real Programmers use Butterflies to edit source code:



When multi-file C programming, there are many source files, eg:



- Module intlist comprising two files (interface intlist.h and implementation intlist.c) - defining a list-of-integers type.
- Separate basic definitions header file defns.h.
- Test program testlist.c, and a main program avgwordlen.c, that use intlists.

So, what should we compile? what should we link?

- What we shouldn't do: gcc -Wall *.c.
- Dependencies between the files are vital, determined by the #include structure. See this via:

```
grep '#include' *.[ch] | grep '"'
```

- Which gives:

```
intlist.c:#include "intlist.h"
avgwordlen.c:#include "intlist.h"
avgwordlen.c:#include "defns.h"
testlist.c:#include "intlist.h"
```

- intlist.c includes intlist.h (to check implementation vs interface).
- avgwordlen.c includes intlist.h (because it uses intlists) and defns.h, etc
- **Make** uses such file dependencies, encoded in a **Makefile**, to automatically compile your programs.
- The Makefile contains dependency rules between target and source files with optional actions (commands) to generate each target from the corresponding sources.

Here's the Makefile:

```
CC      = gcc
CFLAGS  = -Wall
BUILD   = testlist avgwordlen

all:    $(BUILD)
clean:
        /bin/rm -f $(BUILD) *.o core

testlist:       testlist.o intlist.o
avgwordlen:     avgwordlen.o intlist.o
avgwordlen.o:   intlist.h defns.h
testlist.o:     intlist.h
intlist.o:      intlist.h
```

- Makefiles also contain macros, eg `$(CC)` which C compiler to use, `$(CFLAGS)` what C compiler flags etc. Environment variables become macros too, eg `$(HOME)`.
- Note that Make needs very few explicit dependencies and even fewer explicit actions, because it already knows that intlist.o depends on intlist.c, and how to compile .c files.

- Effectively, Make sees the more complete compilation rule:

```
intlist.o:          intlist.c intlist.h
        $(CC) $(CFLAGS) -c intlist.c
```

- This rule declares that intlist.o is up to date only if it is newer than intlist.c and intlist.h. If it doesn't exist or is older than either file, then the action is triggered - compiling intlist.c.
- make takes optional target names on the command line (defaulting to the first target), then performs the minimum number of actions needed to bring the desired targets up to date, based on the timestamps of the target and source files.
- For example, if intlist.h is altered, you run make, that builds the target all, which recursively applies all the rules checking timestamps and concludes that...
- ...intlist.c, testlist.c and avgwordlen.c need recompiling, and then the new testlist.o and avgwordlen.o need relinking against the new intlist.o, giving the 2 executables testlist and avgwordlen.

- If, instead, make is run after intlist.c is modified, it figures out that it needs to recompile intlist.c, and relink both executables against the new intlist.o.
- If, instead, make is run after nothing is modified, it figures out that nothing needs to be done. This parsimonious property of Make is its best feature!
- It's easy to auto-generate Makefiles for single directory C projects containing a single main program and any number of modules - see tarball **02.c-mfbuild** and **03.perl-mfbuild** for two attempts.
- Summary: Always use make, or some similar tool. Keep your Makefile dependencies up to date, optionally auto-generating your Makefiles.
- Google make tutorial for more info.

- As a C project gets larger, you may wish to break it into several sub-directories.
- Core concept: each sub-directory contains:
  - One or more modules (each a paired .c and .h file as usual).
  - Ideally these modules should only depend on each other, and the C standard library.
  - Along with any associated test programs.
  - Plus a Makefile that compiles all the .c files, builds all the test programs, and builds a library containing the .o files belonging to those modules.
- Let's split our existing intlist and avgwordlen directory up.
- What to split? The intlist module is:
  - Logically separate.
  - Reusable - whenever we want a list of integers.
  - Depends on only the standard library.
- That is, it's highly cohesive.
- So: it's perfect for splitting out into a library sub-directory.
- In tarball directory 04.intlist-with-lib, you'll see what we have done to achieve this.

- There's a separate lib sub-directory, let's explore it first:
- lib contains intlist.c, intlist.h, testlist.c and it's own Makefile, lib/Makefile, which builds two core targets:
  - The executable testlist.
  - The library libintlist.a containing intlist.o.
- To do this, lib/Makefile has the following new parts:

```
LIB     =       libintlist.a
LIBOBJS =       intlist.o
BUILD   =       testlist $(LIB)
...
$(LIB):         $(LIBOBJS)
        ar rc $(LIB) $(LIBOBJS)
        ranlib $(LIB)
```

- The new rule says that `$(LIB)` depends on `$(LIBOBJS)`, i.e. `libintlist.a` depends on `intlist.o`, and that the action invokes ar and ranlib - tools that build library files.

- The top-level directory contains avgwordlen.c and defns.h, and a Makefile, containing the following new parts:

```
CFLAGS  =       -Wall -Ilib
LDLIBS  =       -Llib -lintlist
BUILD   =       libs avgwordlen
```

- In CFLAGS, `-Ilib` tells the C compiler to search for include files in the lib directory.
- In LDLIBS, `-Llib` tells the linker to search for libraries in the lib directory, and `-lintlist` links the intlist library in.
- In BUILD, I've added libs before avgwordlen. Later down the main Makefile, we see a rule to make libs:

```
libs:
        cd lib; make
```

- This new always run rule tricks Make, with it's single directory view of the world, into first building in the lib sub-directory, before building in the current directory.

- You'll also notice the new target:

```
spotless:       clean
        cd lib; make clean
```

- I chose a separate spotless target, because in my head, `make spotless` cleans more thoroughly than `make clean`.
- In tarball 05.libintlist and 06.avgwordlen-only, you'll see how to split the intlist module out completely from the avgwordlen application that uses intlists.
- In brief: 05.libintlist contains only the files from the lib directory.
- It's Makefile adds a new install target to install the library into your `~/c-tools/lib/x86_64` directory, and install intlist.h into `~/c-tools/include`.
- After running `make install` in 05.libintlist, your `~/c-tools` library permanently contains the intlist ADT, for you to reuse whenever you like - as shown in 06.avgwordlen-only.
- Left for you to work through!

- I recommend learning Make thoroughly, and personally I find it's all I need.
- But if you find that keeping Makefiles up to date begins to bore you (especially at large scale), there are alternatives - or frontends - to Make:
- For example CMake and the Gnu autoconf system, both of these generate Makefiles automatically from simpler inputs, and are supposed to scale well. Let's briefly look at CMake:
- In tarball 07.intlist-with-cmake you will find a copy of our familiar intlist-with-lib example, in which the only differences are that the Makefiles have been replaced with CMakeLists.txt files, and the README has been modified to explain it.
- Go through that, and you'll get a taste of how CMake lists files are constructed. But CMake is over complex for my tastes. Also, any tool that needs to be run in it's own `build` subdirectory in order to leave the source code directory uncluttered is too messy for me!