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

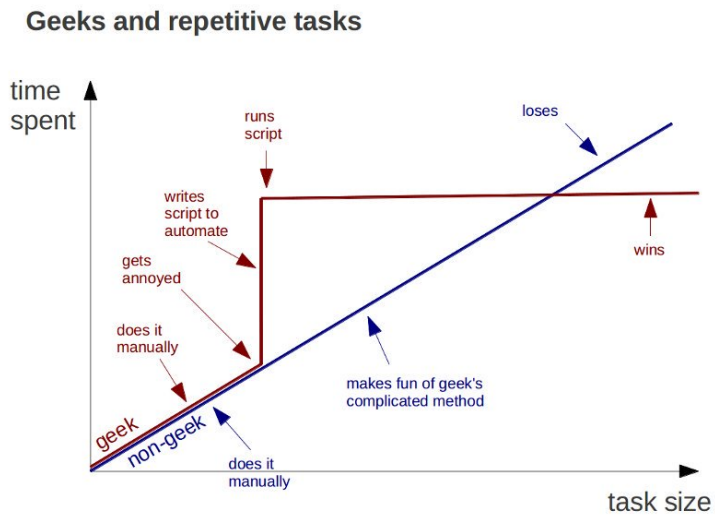
- As programmers, you will learn many languages over your career. Right now, you're learning C - with Will Knottenbelt. C is an excellent language to learn, long-lived (over 40 years old), still going strong, and irreplaceable in it's niches.
- When learning a new language like C, there are several learning stages before you achieve basic competence:
  - Learn the syntax.
  - Learn the basic semantics.
  - Learn the tricky bits eg. pointers and memory management.
  - Learn the standard library (strcpy(), printf(), qsort(), bsearch()..).
  - Learn how to write multi-module programs.
  - Learn the idioms and best practices.
  - Learn to avoid bad practices: the traps and pitfalls.
  - Learn how to write portable code.
- But these Programming Tools lectures are not going to help you achieve basic competence in C.

They are going to attempt to answer the following question:
What comes after that - after you achieve basic C competence?

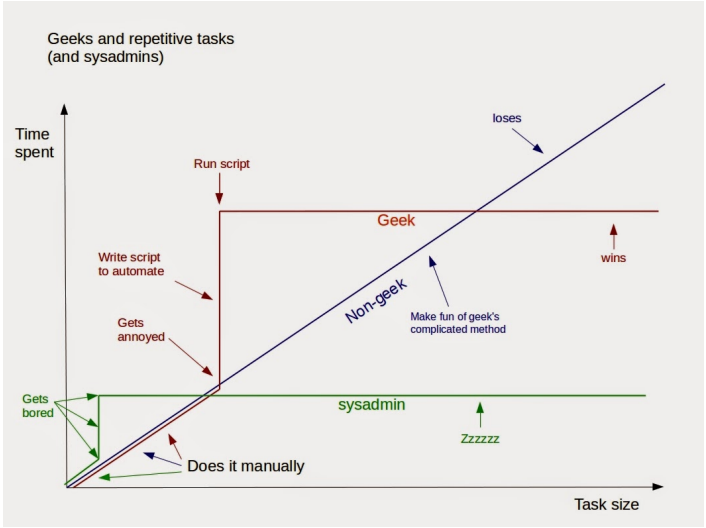
- My answer is: Craftsmanship!
- Build your own toolkit of:
  - Useful tools,
  - Useful libraries,
  - Craft skills to use them.
- To make C programming easier and more productive.
- In fact, to make you look like a master craftsman - an expert C programmer.
- When necessary: don't be afraid to Build your own Tools!
- The Core Principle: Ruthless Automation.
- Doing something boring and repetitive? especially for the second or third time?
- You are a programmer, automation is your core skill - so think: Can I save time by automating this boring task?

Or, to put that another way (as seen on the walkway a couple of years ago):

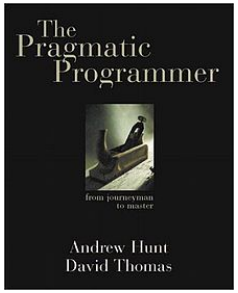
Or, to put that another way (thanks due to SwissMiss):

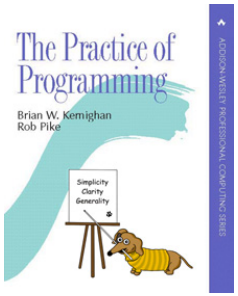


Or, adding SysAdmins into the mix:



For these Programming Tools lectures, I strongly recommend two books:





- The Pragmatic Programmer (PP), by Hunt & Thomas. The carpentry metaphor - and a series of excellent Programming Tips - comes from there.
- The Practice of Programming (PoP), by Kernighan & Pike.

Both books are brilliant expositions of expert-level programming craft.

Contents

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: alternatives to make.

Notes:

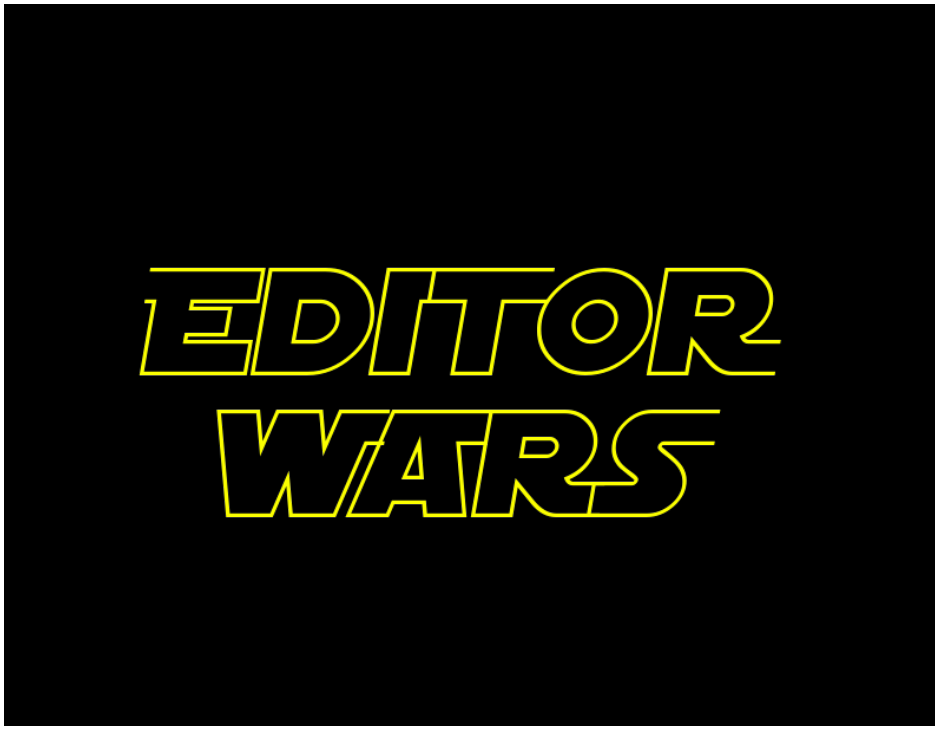
- The handout (these slides) and a tarball (of examples and tools) are available on Materials and `http://www.doc.ic.ac.uk/~dcw/c-tools-2021/lecture1/`
- As a shorthand tarball 01.intlist refers to the directory called **01.intlist** inside the examples tarball. Each directory contains a README file (and indeed there's a top-level README). Read them - and if they tell you to do something, do it right away!

Programmer's Editors

- 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.*

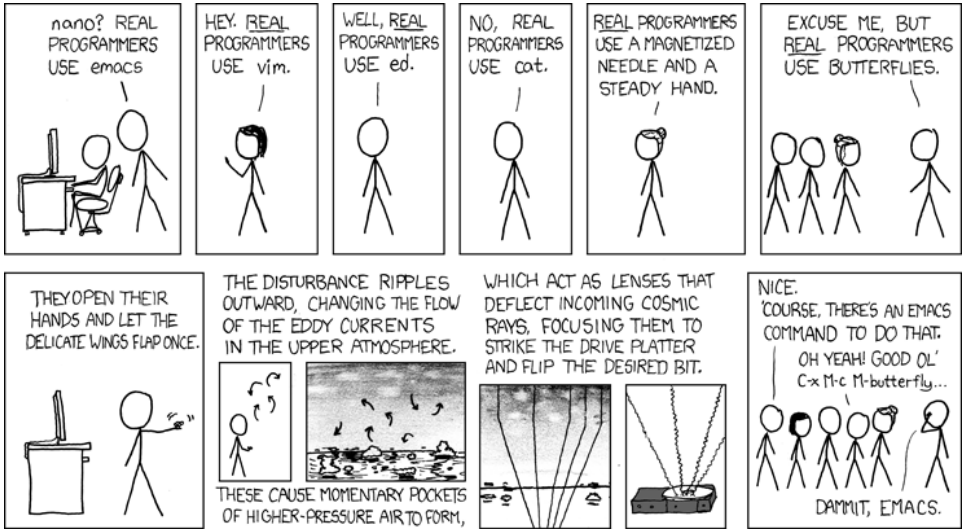
- As a programmer, you will spend years of your life editing programs.
- Coding might be 80% thinking and 20% typing, but your typing must not interfere with your thought process.
- So: Explore several programmer's editors, then choose one, and learn to become expert in it.
- That includes: learning how to plug external tools in.
- Which editor? 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:

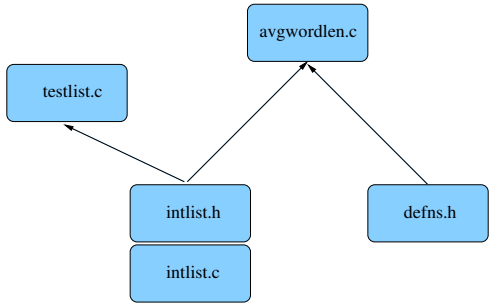
- IDEs such as Idea and CLion provide an editor, an automated compilation system and a debugging environment. If you're going to use an IDE, learn how to use it well, and how to extend and program it.
- Note that Hunt & Thomas aren't much in favour of IDEs. They say "We need to be comfortable beyond the limits imposed by an IDE". Neither am I:-) Neither is Will Knottenbelt.

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



Automatic Compilation

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. A separate basic definitions header file defns.h,
- A unit test program testlist.c, that uses and tests intlist,
- A main program avgwordlen.c, that uses intlists to do something

- So, what should we compile? what should we link?
- What we shouldn't do: gcc -Wall *.c. Why not? Try it out and see!
- Without Make, the correct way to make gcc compile and link everything is:
  1. Tell gcc to compile every .c file onto a corresponding object (.o) file, either all together:
     ```
     gcc -Wall -c *.c
     ```
     or one at a time: foreach .c file:
     ```
     gcc -Wall -c THAT.c
     ```
  2. Then link each main program's .o file with the .o files of all the modules it uses (directly or indirectly), creating a named executable:
     ```
     gcc avgwordlen.o intlist.o -o avgwordlen
     gcc testlist.o intlist.o -o testlist
     ```
- But this is really painful, far too complex for us to do repeatedly.
- We need a tool to handle automatic compilation and linking for us.
- That tool is make. But for it to do the job, we have to tell it the rules: when should each .c file be recompiled, and when should each executable be relinked from it's collection of object files.

- The key information that make will need is related to dependencies between the source files - 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.
- testlist.c includes intlist.h
- **Make** needs such file dependencies, encoded as **Makefile** dependency rules between target and source files with optional actions (commands) to generate each target from the corresponding sources.

- Here's the Makefile for our example. It starts with some variable or macro definitions:
  ```
  CC      = gcc
  CFLAGS  = -Wall
  BUILD   = testlist avgwordlen
  ```
  $(CC) sets which C compiler to use, $(CFLAGS) is the C compiler flags, $(BUILD) the targets to build. Note that environment variables automatically become macros, eg $(HOME) represents your home directory.
- The remainder of the Makefile lists the target/sources/action rules:
  ```
  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
  ```

- Note that Make needs very few explicit dependencies and even fewer explicit actions, because it already knows intlist.o depends on intlist.c, and how to compile .c files.
- So, when you write the rule:

```
intlist.o:      intlist.h
```

(assuming a `intlist.c` file exists in the current directory)
Make expands it to 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 intlist.o doesn't exist or is older than either file, then the action is triggered - compiling intlist.c, producing intlist.o.
- 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...
- ...everything needs to be recompiled and linked.
- 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!
- Note: You have to keep the dependencies in your `Makefile` up to date, otherwise `make` may not know to recompile something.
- If dependency maintenance irritates you, it's surprisingly easy to auto-generate Makefiles for single directory C projects - see tarball **02.c-mfbuild** and **03.perl-mfbuild** for two of my attempts.
- Make continues to work well for any size project - as long as it's all stored in a single directory.

Multi-directory Projects with Make

- As a C project gets larger, you may wish to break it into several sub-directories. Make doesn't handle this natively, but we can handle this with a Makefile per directory and some cleverness.
- Each sub-directory should contain:
  - One or more modules (each a paired .c and .h file as usual).
  - 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 (and it's test program) is the obvious choice. It's:
  - Logically separate - it's highly cohesive.
  - Reusable - whenever we want a list of integers.
  - Depends on only the standard library.

- In tarball directory 04.intlist-with-lib, you'll see what we have done to achieve this:
- There's a separate lib sub-directory to explore, which contains intlist.c, intlist.h, testlist.c (all unmodified) and it's own 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 rcs $(LIB) $(LIBOBJS)
```

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

- The top-level directory contains avgwordlen.c, 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` tells the linker to link 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 tricks Make, with it's single directory view of the world, into first building everything in the lib sub-directory, before building avgwordlen in the current directory.

- You'll also notice the clean target now reads:

```
clean:
        /bin/rm -f $(BUILD) *.o core
        cd lib; make clean
```

which makes it clean in both directories.

- Next: in tarball 05.libintlist and 06.avgwordlen-only, we go a step further: we 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, I've used it for many years with great success.
- But there are alternatives - or frontends - to Make: For example CMake and Gnu autoconf - both generate Makefiles automatically from simpler inputs, and are supposed to scale well.
- 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.
- However: CMake is over complex for my tastes. It generates Makefiles that are thousands of lines long, completely unnecessarily. Also, any tool that needs to be run in it's own `build` subdirectory in order to leave the source code directory uncluttered is badly designed IMO.
- What about Gnu autoconf? It's widely used for portable software distribution, so is very powerful. I have to admit it's still on my

CBuild: A New C-Builder

- Last year, while reviewing the Perl mfbuild program, an idle thought struck me:
- *How hard would it be to take the front end (dependency analysis) part of mfbuild, and instead of generating a Makefile and leaving the compilation and linking to Make, to do that myself. How hard is it to write a simple make-style dependency algorithm?*
- So, I thought I'd have a go at it (clearly, time was hanging heavy on me in the first Lockdown:-)). It turns out to be delightfully simple to do this.
- In tarball 08.cbuild you will find the result. May I present cb - the new C builder. First, go into that directory, look around, and then make install - you now have a new `cb` command, and a man page `man cb` which explains how to use it.

- There are 5 small C projects (in the test1..5 directories) which show off various features of cb - including it's subdirectory support. In particular, you will note that there is no Makefile in any of those directories.
- However, in each directory there is a much simpler `.build` file. Let's go inside `test1`, look around. The `.build` file reads:

```
BUILD   =       avgwordlen testlist
```

- Containing no rules, it looks very like one of the macro declarations from the top of a typical Makefile.. Later on, we'll see that less familiar declarations may be added to assist with multiple-directory work, installation and even testing.
- Now type `cb`. Lo and behold, the source code is compiled, exactly as make would have done. Type `cb` again, and just like make, no compilations are needed - cb is parsimonious too.
- Type `cb clean` and you'll see that it figures out what should be cleaned, all by itself.

- In `test2` you'll see how cb supports building a library, and installing things. It's `.build` file has 3 parts:

```
# Library LIB will be built from LIBOBJS
LIB     =       libintlist.a
LIBOBJS =       intlist.o

# what to build
BUILD   =       testlist $(LIB)

# installations to perform (each mode, source, destination)
INST1   =       644 $(LIB) $(LIBDIR)
INST2   =       644 intlist.h $(INCDIR)
```

- Type `cb` and the test program and the library are compiled and linked. Type `cb install` and the library is installed into `$(LIBDIR)`. But where is LIBDIR set? Look in `../.build` and you'll see. Settings are inherited from the `.build` in the parent directory to save repitition.
- In the remaining test directories - left for you to investigate - you'll see examples of how cb supports libraries in separate subdirectories, and how it handles testing (via `cb test`).
- cb took me a couple of days to write - but of course it was based on the earlier Perl mfbuild so perhaps half the code was already written. It's still experimental, and it may not be feature complete. But I've already started using it as a potential mfbuild+make replacement.
- Give it a try - see what you think of it! Please give me any feedback.