Tools 1

The Core Principle: **Ruthless Automation**

Doing something boring and repetitive, especially for the second or third time?

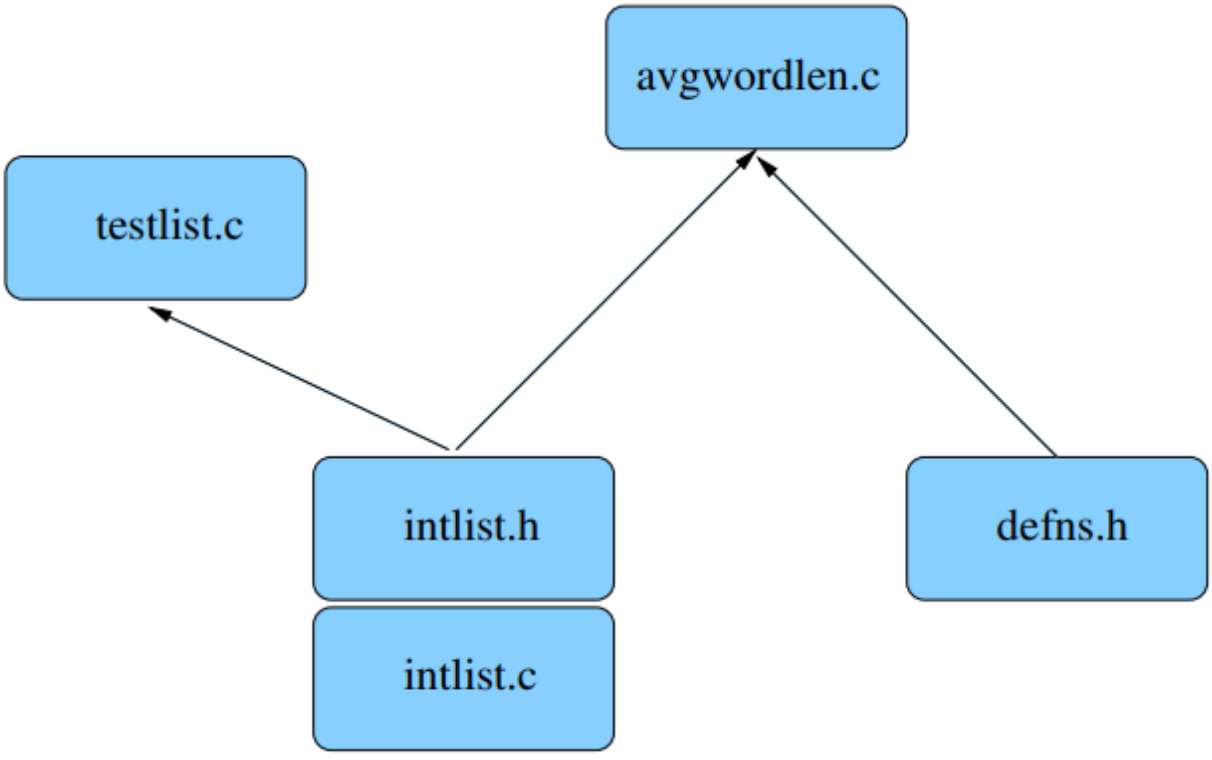
Can I save time by automating this boring task?

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

The leading Programmer's editors are vim and emacs: IDEs such as VSCode 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



How do we compile and link this?

Method 1: `gcc`

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
   ```

This is really painful, far too complex for us to do repeatedly

We need a tool to handle automatic compilation and linking. That tool is make

Method 2: `make`

For it to do the job, we have to tell it the rules:

- When should each .c file be recompiled
- When should each executable be relinked from it's collection of object files

Single Directory Projects with `make`

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

- Note that Make needs very few explicit actions, because it already knows how to compile .c files onto the corresponding .o file, and how to link several .o files together with the C library.
- So, when you write the rule:

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

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.

Method 3: CBuild

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 `.cbuild` file. Let's go inside `test1`, look around. The `.cbuild` 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 `.cbuild` 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 `../.cbuild` and you'll see. Settings are inherited from the `.cbuild` 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.