Building your own C Toolkit

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

Dept of Computing,
Imperial College London

31st May 2012

- When learning any new language, you go through several stages.
- Once you're competent in C - familiar with writing multi-module programs using pointers (malloc() etc) and the standard library (strcpy(), printf(), qsort() etc) - move to the next stage:
- Like a carpenter, build your own toolkit of useful tools to make C programming easier and more productive.
- Sometimes you even need to build your own tools!
- Principle: ruthless automation - when you find yourself doing something boring and repetitive, especially for the second or third time, think: can I automate it?
- Today, I'd like to show you some of the tools in my toolkit, hopefully they'll be useful to you!

We'll cover:

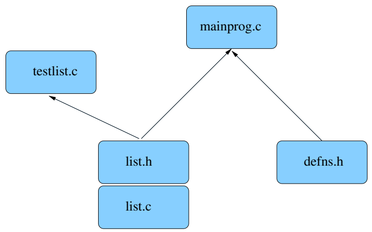
- Basic Tools:
  - Programmer's Editors: Use a single editor well.
  - Automating Compilation (reminder): Use Make.
  - Automating Testing: ruthless testing.
  - Debugging: Use a debugger and know it well.
- Advanced Tools:
  - Generating prototypes automatically.
  - Fixing memory leaks.
  - Optimization and Profiling.
  - Generating ADT modules automatically.
  - Reusable ADT modules: hashes, sets, lists, trees etc.
- I strongly recommend The Pragmatic Programmer (PP) book, by Hunt & Thomas. The woodworking metaphor comes from there.
- There's a tarball of examples associated with this lecture, tarball 01.list refers to a directory inside the tarball. Each directory contains a README file describing what's in it in great detail.

- Hunt & Thomas write:

  > *The editor should be an extension of your hand; make sure your editor is configurable, extensible and programmable.*

- Not my business to tell you which editor to use; avoid editor wars.
- IDEs such as Eclipse provide an editor, an automated compilation system and a debugging environment. If you're going to use an IDE, invest time learning how to use it well, and how to extend and program it.
- I use vi, terse but powerful, extensible in several ways - eg. macros and a "pipe through external command" mechanism.
- Others like Emacs, very powerful and extensible. Like Eclipse, Emacs can be a whole development environment.
- Whichever editor you chose, after initial exploration of the possibilities, stick to it, learn it thoroughly and become expert in its use.

When multi-file C programming, eg:



Many files:

- Module list comprising two files (interface list.h and impln list.c).
- Test program testlist.c
- Main program mainprog.c
- Separate basic defns header file defns.h.

Dependencies between the files are vital, determined by the #include structure:

- list.c includes list.h (check implmn vs interface).
- testlist.c includes list.h
- mainprog.c includes list.h and defns.h

Make uses such file dependencies to automatically compile your programs. Details are covered in another lecture.

- Always use make. Keep your Makefile up to date.
- Exercise: why not auto generate your Makefiles?

- Hunt & Thomas write:

  Tests that run with every build are much more effective than test plans that sit on a shelf.

- Test ruthlessly and automatically by building unit test programs (one per module) plus overall program tests.
- Add make test target to run the tests. Run them frequently.
- Can run make test when you check a new version into git!
- Test programs should check for correct results themselves (essentially, hardcoding the correct answers in them).
- make test could run all test programs in sequence:

```
test:   testprogram1 testprogram2 ...
        ./testprogram1
        ./testprogram2
```

  or invoke a test framework script with testprograms as arguments.
- Exercise: add test target to 01.list to run the obvious `./testlist`, or `./testlist|grep -v ok` to only report failures.
- Test Driven Development (TDD) writes the test programs before implementing the feature to test.

- Suppose your program crashes or produces the wrong answers; you want to debug it. Example in 02.string-debug.
- Choose one debugger and know it well. I recommend gdb, the GNU debugger, which works with C++ too:
- Recompile all source code with gcc flag -g: set `CFLAGS = -Wall -g` in your Makefile, then recompile everything via `make clean all`.
- Start gdb by `gdb PROGRAMNAME`. Inside gdb, type `run COMMANDLINEARGS`. Work with your program until it crashes.
- Back at the gdb prompt: type `where` to see the call frame stack - the sequence of function calls leading to the crash.
- `frame N` allows you to switch to the Nth function call on the frame stack, i.e. select which of the function calls you want to look at, in order to examine that function's local variables.

- `list` will list 10 lines of the current function.
- `p EXPR` will print any C expression, including global variables and local variables in the current stack frame.
- `whatis VAR` displays the type of VAR.
- `x` is a flexible memory dumper. `x/12c &str` would print out the first 12 bytes of data from `str` in ASCII, `12xb` as hexadecimal etc. `help x` (inside gdb) for more info.
- You can also set breakpoints (`break LINENO|FUNCTIONNAME`), attach conditions on the breakpoints, single step through your program (`step` and `next`), continue until you hit another breakpoint (`cont`), and even watch variables as they are altered or accessed (`watch, rwatch`).
- Google for gdb tutorial for more info.
- Most important, leave gdb by `quit`.

- Irritating C problem: keeping the prototype declarations in interfaces (.h files) in sync with the function definitions in the implementation (.c files).
- Whenever you add a public function to list.c you need to remember to add the corresponding prototype to list.h.
- Even adding or removing parameters to existing functions means you need to make a corresponding change in the prototype too.
- Don't live with broken windows (PP tip 4) - write a tool to do the work, then integrate it into your editor for convenience!
- Years ago, I wrote proto - a tool to solve this. It reads a C file looking for function definitions, and produces a prototype for each function. LIMITATION: whole function heading on one line.
- Then I wrote a vi macro bound to an unused key that piped the next paragraph into proto % (current filename). Can do same for forward declarations of static functions using proto -s %.

Memory leaks are the most serious C problem:

- Often claimed that 99% of serious C bugs are memory-allocation related. C uses pointers and `malloc()` so much, with so little checking, that debugging memory related problems can be challenging even with gdb.
- Failing to free() what you malloc() is very bad for long running programs, that continuously modify their data structures.
- free()ing a block twice is equally dangerous.
- derefencing an uninitialized/reclaimed pointer gives non-deterministic behaviour (really hard to debug!).
- Segmentation faults - gdb where (frame stack) may show it crashes in system libraries.

- Why can't the system diagnose these?
- There are several tools that can - Electric Fence and valgrind/memcheck among them.
- Here's a homebrew alternative: the August 1990 Dr Dobbs Journal provided libmem, a very simple C module which uses the C pre-processor to redefine malloc(), free(), exit() etc to add extra checking.
- Let's see it in action:
  - First install libmem from tarball directory 04.libmem
  - Now go into tarball directory 05.mem-eg, 2 test programs.
  - `make` and run the programs without libmem.
  - Add `#include <mem.h>` to both .c files
  - Add `-lmem` to `LDLIBS` in Makefile
  - Rebuild using `make clean all`
  - Run the two examples now!

- Suppose we have a pre-written, pre-tested hash table module. Passes all tests (creating, populating, finding, iterating over, freeing a single hash table). Pretty confident that it works!
- But we haven't checked it with libmem yet!
- When we embed it in a larger system, we'll create, populate and destroy whole hash tables thousands of times.
- Voice of bitter experience: Test that scenario before doing it:-)
- New test program `iterate N M` that (silently) performs all previous tests N times, sleeping M seconds afterwards.
- Behaviour should be linear with N. Test it with `time ./iterate N 0` for several values of N, graph results.
- Find dramatic non-linear behaviour around 6-7k iterations on some older lab machines: Twice as slow, CPU %age falls, starts doing I/O.
- What's happening?

- Try monitoring with top, configured to update every minute (d 1), sort by %age of memory (O n). Write this config out (W).
- Run iterate with a time delay: `time ./iterate 8000 10` and watch top! iterate's memory grows bigger than the physical memory, tops out at about 85% of physical memory, the system starts swapping (%wait goes busy), machine goes very slow!
- A job for libmem: Find the details in 07.badhash+mem's `README`, but in summary libmem enables us to track down a missing free() pretty easily.
- Conclusion: compile everything with libmem from day one. Save yourself loads of grief, double your confidence.
- Exercise: verify that the list example (in 01.list) runs cleanly with libmem. (Import CFLAGS and LDLIBS from 05.mem-eg's Makefile).

- gcc and most other C compilers can be asked to optimize the code they generate, gcc's option for this is -O. This is worth trying, but doesn't often make a significant difference.
- What makes far more difference is finding the hot spots using a profiler and selectively optimizing the hot spots. Can produce dramatic speedups, and profiling often produces surprises.
- Compile and link with -pg, which generates instrumented code which, when run, produces a binary profiling data file. The tool gprof then relates the data file to the executable and produces a report showing the top 10 functions (across all their calls) sorted by percentage of total runtime.
- Let's try profiling the bugfixed hash module's iterate test program, and see what surprises there may be.

- Profiling iterate 10000 gives the following table:

```
  %    cumul    self              self   total
 time  seconds  seconds   calls us/call us/call name
 38.71   3.37   3.37      20000  168.37  206.96 hashFree
 22.92   5.36   1.99      10000  199.44  289.14 hashCopy
 11.29   6.34   0.98      10000   98.22   98.22 hashCreate
 10.31   7.24   0.90 325330000   0.00     0.00 copy_tree
  8.87   8.01   0.77 650660000   0.00     0.00 free_tree
```

- 650 million calls to free_tree and 325 million calls to copy_tree are highly suspicious. Aha! The hash table's array of trees has 32533 entries!
- hashFree and hashCopy have the same structure, iterating over the array of trees making one call to free_tree/copy_tree per tree. The vast majority of these trees are empty.
- We double the speed of iterate by adding if(the_tree != NULL) conditions on tree calls in hashFree, hashCopy and others.
- We might also consider shrinking the size of the array of trees to some smaller prime number - or, more radically, adding code to dynamically resize the array (and rehash all the keys?) while in flight.

- Principle: It's often an excellent idea to import cool features from other languages.
- For example, Perl teaches us the importance of hashes (aka Java dictionaries) - (key,value) storage implemented using hash tables. We've already seen a hash module bring this ability to C.
- Many years ago, I realised that one of the best features of functional programming languages such as Haskell is the ability to define recursive shaped data types, as in:

  `intlist = nil or cons( int head, intlist tail );`
- I'd dearly love to have that ability in C. If only there was a tool that reads such type definitions and automatically writes a C module that implements them..
- I looked around, couldn't find anything anywhere. Noone but me seemed to have ever thought that such a tool might even be useful!

- So I wrote one! A week or two's work one summer, the result was datadec - in the 09.datadec directory, also installed on DoC linux machines. After installing it, use it as follows:
- In 10.datadec-eg you'll find an input file types.in containing:

```
TYPE {
        intlist =  nil or cons( int first, intlist next );
        illist  =  nil or cons( intlist first, illist next );
        idtree  =  leaf( string id )
                or node( idtree left, idtree right );
}
```

- To generate a C module called datatypes from types.in, invoke:

```
datadec datatypes types.in
```

- datatypes.c and datatypes.h are normal C files, write test programs against their interfaces, use them. Don't modify them!
- But you can modify the input file - suppose you realise that an idtree leaf needs two strings not one. Simply change the type defn and rerun datadec. Now the `idtree_leaf()` constructor takes two arguments not one!

- Whether generated by datadec or written by hand, most problems are made a lot easier by a library of trusted modules:
    - indefinite length dynamic strings
    - indefinite length dynamic arrays
    - linked lists (single or double linked)
    - queues and priority queues
    - binary trees
    - hashes
    - sets - hashes with no values? trees? sparse arrays?
    - bags - frequency hashes
    - anything else you find useful (.ini file parsers? test frameworks?)
- The C standard library fails to provide any of these (C++ provides the Standard Template Library of course).
- So build them yourself as and when you need them, and reuse them at every opportunity, to raise C to a higher level!
- Reuse can be done without object orientation, it's not hard!

- Grow your C skills by building a powerful toolkit that makes C programming simpler.
- Choose tools you like; become an expert in each one.
- When necessary, build tools yourself. Don't be afraid!
- I didn't mention: lexical analysers (lex/flex), parser generators (yacc/bison); regular expression libraries; all the things you can do with function pointers; defining little languages; text processing tools; OO programming in C etc etc.
- These slides and the C-tools tarball are available at

  `http://www.doc.ic.ac.uk/~dcw/c-tools/`
- Most importantly: enjoy your C programming! Build your toolbox - and let me know if you write any particularly cool tools!