Building your own C Toolkit: Part 2

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

Dept of Computing,
Imperial College London

5th June 2014

- Last week, we introduced the idea of building a C programming toolkit, and covered the following tools or techniques:
  - Programmer's Editors.
  - Automatic Compilation: Make.
  - Automatic Ruthless Testing.
  - Debugging: gdb.
  - Building shortlived tools on the fly.
- Today, we're going to carry on, and cover:
  - Generating prototypes automatically: proto.
  - Fixing memory leaks: libmem.
  - Optimization and Profiling.
  - Generating ADT modules automatically.
  - Reusable ADT modules: hashes, sets, lists, trees etc.
- As last week, there's a tarball of examples associated with this lecture. Both lectures' slides and tarballs are available on CATE and at: `http://www.doc.ic.ac.uk/~dcw/c-tools-2014/`

- 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 must be placed 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.
- Such programs can 'leak' memory until they run out of memory (use more memory than the computer has physical RAM)!
- free()ing a block twice is equally dangerous.
- dereferencing 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(), strdup() etc to add extra checking.
- Let's see it in action:
  - First `make install` libmem from tarball directory 02.libmem
  - Now go into tarball directory 03.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! They tell you exactly what you've forgotten to `free()`! Magic!
- You may say: but those test programs are tiny. Does libmem scale to larger size programs?

- Suppose we have a pre-written, pre-tested hash table module, plus a unit test program **testhash**. Passes all tests (creating, populating, finding, iterating over, freeing a single hash table).
- We've even used it in several successful projects - so we're pretty confident that it works!
- But we have never checked it with libmem! Why not?
- When we prepare to embed our hash table in a larger system, we'll need to 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 (with M=0) 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 10-11k iterations on lab machines: Twice as slow, CPU %age falls, starts doing I/O.
- What on earth is happening?

- Try monitoring with `top`, configured to update every second (d 1), sort by %age of memory (O n). Write this config out (W).
- Run iterate with a time delay: `time ./iterate 11000 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), load average goes high, machine goes very slow!
- Hypothesis: the hash table module is leaking some memory, ie. failing to free everything that it mallocs. A job for libmem!
- Proceed as before:
  - append -lmem to LDLIBS in the Makefile
  - edit *.c and add `#include <mem.h>` to each
  - rebuild using 'make clean all'
  - run `./testhash` [simpler test program]
  - result: 2 non-freed 256K chunks reported:

```
File     Line   Size
hash.c   114    260264
hash.c   75     260264
```


- Libmem debugging session continued:
  - look at those two lines: line 75 is in `hashCreate(...)`:

    ```
    h->data = (tree *) malloc( NHASH*sizeof(tree) );
    ```
  - and line 114 is nearly identical in hashCopy().

    ```
    result->data = (tree *) malloc( NHASH*sizeof(tree) );
    ```
  - Look in corresponding `hashFree(hash h)` function.
  - Aha! `h->data` is NOT FREED.
  - Add the missing `free(h->data)`, recompile (make).
  - Rerun `./testhash` and it reports no unfree()d blocks.
  - Rerun `./iterate 11000 10` again - no non linear behaviour, no memory leak reported. Job done! libmem rocks!
- Summary: compile everything with libmem from day one. Save yourself loads of grief, double your confidence.
- Exercise: verify that the list example (in Lecture 1's 01.list) runs cleanly with libmem. (Import CFLAGS and LDLIBS from 03.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. Worth trying, rarely makes a significant difference.
- What makes far more difference is finding the hot spots using a profiler and selectively optimizing them. Can produce dramatic speedups, and profiling often produces surprises.
- Let's try profiling the bugfixed hash module's iterate 10000 test program, and see what surprises there may be:
  - Add -pg to CFLAGS and LDLIBS in Makefile.
  - Run `make clean all` (compile and link with -pg, which generates instrumented code which tracks function entry and exit times.
  - Run `./iterate 10000`, which runs a bit slower than normal (because profiling slows it down a bit), producing a binary profiling file called gmon.out.
  - The tool gprof then analyzes the executable and the data file, producing a report showing the top 10 functions (across all their calls) sorted by percentage of total runtime. Run:
    `gprof ./iterate gmon.out > profile.orig`

- head profile.orig shows results like:

```
 %     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 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 can 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) when the hash gets full.

- 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 inductive 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 seemed to have ever suggested that such a tool could be useful!
- Decision time: do I abandon my brilliant idea, or make the tool?
- Think hard: a serious tool, parser, lexical analyser, data structures, tree walking code generator: at least a week's work!

- I made the tool! After a fortnight's work, the result was datadec - in the 08.datadec directory, older version installed on DoC linux machines. After installing it, use it as follows:
- In 09.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, you can read them, write test programs against the interface, use them in production code. But don't modify these files - if you do then you can't...
- ... change types.in later - suppose you realise that an idtree node needs to store an id as well as the trees. Change the type defn, rerun datadec. The `idtree_node()` constructor now takes 3 arguments!

- Let's look inside datatypes.h, to find what idtree functions datadec generates. First we find two *constructors*:

```
extern idtree idtree_leaf( string );
extern idtree idtree_node( idtree, idtree );
```

- Then we find a function telling you whether a tree is a leaf or a node:

```
extern kind_of_idtree idtree_kind( idtree );
```

  Using the enumerated type:

```
typedef enum { idtree_is_leaf, idtree_is_node } kind_of_idtree;
```

- Then two deconstructor functions which, given a tree of the appropriate shape, breaks it into it's constituent pieces:

```
extern void get_idtree_leaf( idtree, string * );
extern void get_idtree_node( idtree, idtree *, idtree * );
```

- The final function prints a tree to a file in human readable format (which you can control):

```
extern void print_idtree( FILE *, idtree );
```

- By default, there's no free functions. Surprisingly hard to automatically generate due to shallow vs deep considerations.
- New this year: run datadec -f.. and get experimental free_TYPE() functions. If you don't want a parameter freed, mark it in the input file with a '-', as in:

```
idtree  =  leaf( -string id )
        or node( idtree left, idtree right );
```


- Looking in testidtree.c, we build two leaves, and then test that we can break them apart again:

```
idtree t1 = idtree_leaf( "absolutely" );
testleaf( t1, "absolutely", "ab" );
idtree t2 = idtree_leaf( "fabulous" );
testleaf( t2, "fabulous", "fab" );
```

- testleaf(t, expected, treename) tests that t is a leaf with the expected id, treename is a symbolic name for the tree:

```
void testleaf( idtree t, char *expected, char *treename )
{
  char label[1024];
  sprintf( label, "isnode(%s)", treename );
  inteqtest( idtree_kind(t), idtree_is_leaf, label );
  string id;
  get_idtree_leaf( t, &id );
  sprintf( label, "getleaf(%s)", treename );
  streqtest( id, expected, label );
}
```

- inteqtest(value, expected, label) and streqtest(value, expected, label) are integer and string equality tests that print ok/fail messages.
- Next, testidtree.c constructs a node from our two leaves, and tests that we can break it apart correctly:

```
idtree t = idtree_node( t1, t2 );
inteqtest( idtree_kind(t), idtree_is_node,
           "isnode((ab,fab))" );
idtree l, r;
get_idtree_node( t, &l, &r );
testleaf( l, "absolutely", "left((ab,fab))" );
testleaf( r, "fabulous", "right((ab,fab))" );
```


- Most problems are made a lot easier by having a library of trusted modules - whether datadec-generated or handwritten:
  - indefinite length dynamic strings
  - indefinite length dynamic arrays
  - indefinite length sparse dynamic arrays
  - linked lists (single or double linked)
  - stacks (can just use lists)
  - 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 the following (C++ provides the Standard Template Library): 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!