Building your own C Toolkit: Part 2

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

Dept of Computing,
Imperial College London

6th June 2013

- 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.
  - Generating prototypes automatically: proto.
  - Fixing memory leaks: libmem.
- Today, we're going to carry on, and cover:
  - Optimization and Profiling.
  - Generating ADT modules automatically.
  - Reusable ADT modules: hashes, sets, lists, trees etc.
  - Building shortlived tools on the fly.
  - Parser and Lexer Generator tools: yacc and lex.
- 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/`

- 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 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 seemed to have ever suggested that such a tool could be useful!
- Decision time: do I abandon my brilliant idea, or write the tool?

- I wrote the tool! A fortnight's work one summer, the result was datadec - in the 03.datadec directory, also installed on DoC linux machines. After installing it, use it as follows:
- In 04.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 them, because then you can't...
- ... modify types.in - suppose you realise that an idtree node needs to store an id as well as the trees. Change the type defn, rerun datadec. Now the `idtree_node()` constructor 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 );
```

- Note that there's no free function. Surprisingly hard to automatically generate - should you free the 'id' parameter inside a leaf or not?
- I recommend the following: while experimenting with types.in, forget free()ing. When your recursive types have become stable, you should write the tree-traversing void free_TYPE(TYPE t) functions yourself. Add them to the GLOBAL section (after @@) in types.in - man datadec for more details.

- 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 a library of trusted modules - provided, generated by datadec 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!

- We often find ourselves writing hundreds of repetitive "pattern instances", differing only in small details, eg:

```
int plus( int a, int b ) { return (a+b); }
int minus( int a, int b ) { return (a-b); }
int times( int a, int b ) { return (a*b); }
...
```

- All that varies from line to line is (funcname,operator), plus perhaps the type 'int' is also a parameter?
- Why not generate them automatically using an ad-hoc tool, scaffolding that you build in 30 minutes or less, use a few times, then discard?
- Specify input format (as a little language) and corresponding output:

```
INPUT:
  line 1: typename T eg. int
  foreach line>1: F, Op pairs

OUTPUT:
  foreach line>1: "T F( T a, T b ) { return (a Op b); }"
```

- Ok, first observe that this is a simple job for a scripting language like Perl, here's a Perl oneliner I composed in about two minutes:

```
perl -nle 'if($.==1){$t=$_;next} ($f,$op)=split(/,/);...
           print "$t $f( $t a, $t b ) { return (a $op b); }"'
```

- Even if we write this in C, might take about 30 minutes using low-level string manipulation, or 10-15 minutes using standard library function strtok(). See 05.tiny-tool/README for details.

- Scaling the previous idea of little languages up, you often need to write parsers and lexical analysers. This problem has been solved! Like datadec, lex and yacc generate C code from declarative definitions of tokens and language grammars.
- As a simple example, consider integer constant expressions such as `3*(10+16*(123/3) mod 7)`. The basic 'tokens' needed are:
  - Numeric constants (eg '123').
  - Various one-character operators (eg. '(', '+', '*', ')' etc).
  - A Haskell-inspired keyword 'mod' (i.e. modulus, '%' in C terms).
- Specify the input tokens as regular expressions:

```
[0-9]+              return NUMBER;
\+                  return PLUS;
-                   return MINUS;
\*                  return MUL;
\/                  return DIV;
mod                 return MOD;
\(                  return OPEN;
\)                  return CLOSE;
\n                  /* ignore end of line */;
[ \t]+              /* ignore whitespace */;
.                   return TOKERR;
```

- See lexer.l for the full lex input file, containing the above rules and some prelude. This file can be turned into C code via: lex -o lexer.c lexer.l.

- These tokens can be combined to form expressions using the following BNF-style grammar rules (in yacc-format):

```
%token PLUS MINUS MUL DIV MOD OPEN CLOSE TOKERR
%token NUMBER

%start oneexpr
%%
oneexpr     :  expr
            ;
expr        : expr PLUS term
            | expr MINUS term
            | term
            ;
term        : term MUL factor
            | term DIV factor
            | term MOD factor
            | factor
            ;
factor      : NUMBER
            | OPEN expr CLOSE
            ;
```

- parser.y contains these rules plus some yacc-specific prelude, including a short main program that calls the parser. This can be turned into C code (parser.c and parser.h) via: yacc -vd -o parser.c parser.y
- You can now compile and link parser.c and lexer.c to form expr1, just type make. See the Makefile for details. expr1 is a recognizer: it will say whether or not the expression (on standard input) is valid.

- Directory 07.expr2 extends our recognizer so that it calculates the value of the expression and displays it. There are two sets of changes from the previous version:
- First, we modify one line in lexer.l to store the value of the integer constant into 'yylval.n':

```
[0-9]+                  yylval.n=atoi(yytext); return NUMBER;
```

- Second, in parser.y there are several changes: add to the prelude:

```
static int expr_result = 0;
```

  Then make main display the result after a successful parse:

```
printf( "result: %d\n", expr_result );
```

- Above the token definitions, add:

```
%union { int n; }
%token <n> NUMBER
%type <n> expr term factor
```

- Add actions to grammar rules with more than one sub-part, taking the calculated value from each sub-part and computing the result, plus a top level action which sets expr_result. Here's a sample:

```
oneexpr     : expr             { expr_result = $1; }
            ;
expr        : expr PLUS term   { $$ = $1 + $3; }
            | expr MINUS term  { $$ = $1 - $3; }
            | term
            ;
term        : term MUL factor  { $$ = $1 * $3; }
            | term DIV factor  { $$ = $1 / $3; }
            ...
```

- After make we have expr2, an expression calculator. Play with it.

- Directory 08.expr3 extends our calculator, allowing a factor to be an identifier - an IDENT token, representing a named constant. There are three sets of changes from the previous version:
- Add a new consthash module, which stores our named constants.
- Add a line in lexer.l to recognise and return our new token:

```
[a-z][a-z0-9]*          yylval.s=strdup(yytext);return IDENT;
```

- parser.y has several changes: add to the prelude:

```
#include "consthash.h"
```

  Then main needs to create the constant hash right at the start, destroy it at the end:

```
init_consthash( argc > 1 );
if( yyparse()....
destroy_consthash();
```

- Change the union declaration to:

```
%union { int n; char *s; }
```

- Tell the parser that IDENT builds a string:

```
%token <s> IDENT
```

- Add the new factor rule:

```
  | IDENT           { $$ = lookup_const($1); }
```

- After make we have expr3, a calculator with named constants. Play with it.

- Directory 10.expr5 contains our final yacc/lex example, which replaces calculation with treebuilding (using datadec): prepare types.in, add Makefile rules:

```
TYPE {
  arithop =  plus or minus or times or divide or mod;
  expr   =  num( int n )
        or id( string s )
        or binop( expr l, arithop op, expr r )
        ;
}
```

- parser.y has several changes: add to the prelude:

```
#include "types.h"
```

- Change expr_result from an int to an expr:

```
static expr expr_result = NULL;
```

- main should print out the expression tree (on parse success):

```
print_expr( stdout, expr_result );
```

- Change the union declaration to:

```
%union { int n; char *s; expr e; }
```

- Change the type of all expression rules to e, the union's expr:

```
%type <e> expr term factor
```

- Change all the actions, for example:

```
expr        : expr PLUS term  { $$ = expr_binop( $1, arithop_plus(), $3 ); }
            | expr MINUS term { $$ = expr_binop( $1, arithop_minus(), $3 ); }
      ...
factor      : NUMBER          { $$ = expr_num($1); }
            | IDENT           { $$ = expr_id($1); }
```

- After make we have expr5, an expression parser and treebuilder.

- If you're not impressed by expression parsers, the tarball also contains 11.haskell-tiny-treebuilder, which defines a tiny Haskell subset, builds a parser, and builds trees to represent it. Still not impressed? 12.haskell-tiny-codegen translates it to C!
- Follow 100,000 years of human history by tool-using and tool-making. Build yourself a powerful toolkit. Choose tools you like; become expert in each.
- When necessary, build tools yourself to solve problems that irritate you. Don't be afraid! Try to build tools that save you more time than they cost you to make.
- I didn't mention: regular expression libraries; all the things you can do with function pointers; text processing tools; OO programming in C etc etc.
- Most importantly: enjoy your C programming! Build your toolkit - and let me know if you write any particularly cool tools!
- Finally, scripting languages like Perl or Python are fantastic timesavers. I run a Perl course each December, notes available at: `http://www.doc.ic.ac.uk/~dcw/perl2012/`