

C Programming Tools: Part 3

Building your own Tools

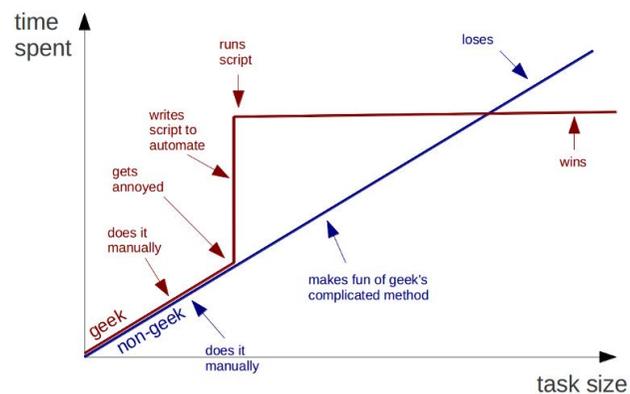
Duncan C. White
d.white@imperial.ac.uk

Dept of Computing,
Imperial College London

June 2018

- The Pragmatic Programmers exhort us to: **Learn a Text Manipulation Language** (tip 28) - such as Perl - and **Write Code that Writes Code** (tip 29).
- Let's see an example of those tips together, remembering..

Geeks and repetitive tasks



- So far, most tools we've covered have already existed (apart from the two Makefile builders we mentioned in passing in Lecture 1).
- But we said then: **When necessary: build your own tools!**

Today, we're going to cover building tools at a range of scales:

- Tiny: Building **shortlived tools on the fly**.
- Medium: **Generating prototypes automatically: proto**.
- Large: **Reusable ADT modules**: hashes, sets, lists, trees etc.
- Large: **Generating ADT modules automatically**.

As in previous weeks, there's a tarball of examples associated with this lecture.

- This lecture's slides and tarballs are available on CATE under Programming III.
- Also at: <http://www.doc.ic.ac.uk/~dcw/c-tools-2018/>

- Suppose we find ourselves writing hundreds of repetitive "pattern instances" like this:

```
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); }
...
```

- If we need to write 10 of them - do it in your favourite **programmer's editor** using clone-and-alter.
- What if we need to write 50 of them? Or 100 of them? Or 100 int functions and another 100 double functions?
- Are we bored yet? Is clone-and-alter too error-prone? Then why not..
- Generate such function instances automatically using a **shortlived tool**, scaffolding that you **build** on demand, **use** a few times, then **discard**:
- Clearly, all that varies from instance to instance is (funcname,operator), eg. (plus,+).
- Specify input format (as a **little language**) and corresponding output:

```
INPUT:
foreach line: F, Op pairs
OUTPUT:
foreach line: "int <F>( int a, int b ) { return (a <Op> b); }"
```

- Simple job for a scripting language like [Perl](#).
- Here's a Perl oneliner I composed in a minute:


```
perl -nle '($f,$op)=split(/,/); print "int ${f}( int a, int b ) { return (a ${op} b); }"' < input
```
- The basic structure:


```
perl -nle 'PERL CODE' < input
```

 means execute that chunk of Perl code for every line of the input.
- The Perl code:


```
($f,$op)=split(/,/)
```

 means split the current line on "," into two strings, storing the part before the comma into the variable \$f, and the part after the comma into \$op.
- The Perl code:


```
print "int ${f}( int a, int b ) { return (a ${op} b); }"
```

 means print out the string literal, replacing \${f} and \${op} with the value of those variables.
- Don't want to do it in Perl? (weirdo). Then use a different tool!
- I wrote it in C in 15 minutes using standard library function `strtok()` to split on comma: See [01.tiny-tool/genfuncs1.c](#).

- Note that our tool doesn't have to be perfect; just good enough to save us time.
- Once you have a tiny tool, don't be afraid to modify it:
- Left-justify the function names in a field of some suitable width:


```
perl -nle '($f,$op)=split(/,/); printf "int %-15s( int a, int b ) { return (a${op}b); }\n", $f' < input
```
- Or, prefix the typename onto function names, eg. `int_plus`:


```
perl -nle '($f,$op)=split(/,/); printf "int %-15s( int a, int b ) { return (a${op}b); }\n", "int_${f}"' < input
```
- Why not let the user change the type at any point in the input:


```
TYPE,int
plus,+
minus,-
TYPE,double
plus,+
minus,-
```

 generates:


```
int int_plus ( int a, int b ) { return (a+b); }
int int_minus ( int a, int b ) { return (a-b); }
double double_plus ( double a, double b ) { return (a+b); }
double double_minus ( double a, double b ) { return (a-b); }
```

- To implement this, change the specification to:


```
INPUT:
  foreach line: F, Op pair (but F=="TYPE" is special)
OUTPUT:
  foreach F, Op pair
    if F=="TYPE" then T=Op
    else print "<T> <T><F>(<T> a, <T> b ) { return (a <Op> b); }"
```
- Make our Perl one-liner:


```
perl -nle '($f,$op)=split(/,/); if( $f eq "TYPE" ) { $t=$op; next; }
  printf "${f} %-15s( ${t} a, ${t} b ) { return (a${op}b); }\n", "${f}_${f}"' < input
```
- Final thought, instead of hardcoding the output format in the printf, we could replace TYPEs with TEMPLATEs in the input, for example:


```
TEMPLATE,int int_<0>( int a, int b ) { return (a<1>b); }
plus,+
minus,-
TEMPLATE,double double_<0>( double a, double b ) { return (a<1>b); }
plus,+
minus,-
```
- Here, the marker <0> means "replace this marker with the current value of the first field". Our Perl one-liner becomes more powerful but shorter:


```
perl -nle 'of=split(/,/,,$_2); if( $f[0] eq "TEMPLATE" ) { $t=$f[1]; next; }
  $_=$t; s/<(\d+)/${f[1]}/g; print' < input
```
- This is now a simple macro processor.

- Let's move on to an example medium scale tool I built.
- While developing C code, you may find certain things irritate you.
- The Pragmatic Programmers describe such things as [broken windows](#), and tell us - in tip 4 - [Don't live with broken windows](#). Find a way to fix the problem!
- One particular thing irritated me some years ago: keeping the [prototype declarations](#) in .h files in sync with the [function definitions](#) in the paired .c files that form modules.
- Whenever you [add a public function](#) to `intlist.c` you need to remember to add the corresponding prototype to `intlist.h`.
- Even [adding or removing parameters](#) to existing functions means you need to make a corresponding change in the prototype too. What a pain!
- The problem here is that there's a lot of repetition between the .c file and the .h file. This violates the most important Pragmatic Programmers tip: [DRY - Don't Repeat Yourself](#) (tip 11).

- Don't live with broken windows suggests we should find, or write, a tool to solve this problem, then integrate it into our editor for convenience!
- Years ago, I wrote `proto` to solve this. It reads a C file looking for function definitions, and produces a prototype (an extern declaration) for each function.
- But this sounds pretty hard. Don't we need a complete C parser?
- I found an easier way. I imposed LIMITATIONS on my layout approach to make the tool easier to construct: I decided that the whole function heading must be placed on one line, and also that the function heading could only use simple type declarations (use `typedef` for complex declarations).
- Then I wrote a vi macro bound to an unused key that piped the next paragraph into `proto %` (current filename). See <http://www.doc.ic.ac.uk/~dcw/PSD/article4/> for an article I wrote about how easy similar editor extensions can be.

- Most problems are made a lot easier by having a library of trusted reusable ADT modules:
 - indefinite length `dynamic strings`
 - indefinite length `dynamic arrays`
 - `linked lists` (single or double linked)
 - `stacks` (can just use lists)
 - `queues` and `priority queues`
 - `binary trees`
 - `hashes` (aka maps/dictionaries/associative arrays).
 - `sets` of strings - several possible implementations.
 - `bags` - frequency hashes, mapping strings to integers.
 - anything else you find useful (.ini file parsers? test frameworks? CSV splitters?)
- Unlike C++, the C standard library fails to provide any of the following: So, either find a collection of such modules that others have written, or `build them yourself` as and when you need them, and `reuse them` at every opportunity.
- Note: Reuse can be done without OO or generics, *Make it Easy to Reuse* (PP Tip 12) - just use `void *`.

- To get you started, `tarball 03.adts` includes a group of half a dozen ADTs (plus unit test programs) that I've written over the years, plus a Makefile to package them as the `libADTs.a` library.
- Investigate them all at your own leisure - but `make install` them now so they're installed in your `TOOLDIR` (`~/c-tools`) directory.
- Next, `tarball 04.hash-set.eg` contains an example application that uses some of those ADTs, specifically:
 - `Hashes` - (`key,value`) storage implemented using hash tables, where the keys are strings, and the values are generic `void *` pointers - yes, it's our old friend `hash.c`, after Lecture 2's memory-leak fixes and profiling-led optimizations.
 - and `Sets of strings`.
 - Then combines them to represent family information, i.e. a mapping from a `named parent` to `set of named children`.
 - It's left for you to examine and play with.
- `C+hashes+sets` makes it easy to pretend that you're almost programming in Perl:-)

- **Principle:** It's often an excellent idea to `import cool features from other languages`.
- 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, *but I couldn't find one*. 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?`
- Cost/benefit analysis: a serious tool, a mini-compiler (with parser, lexical analyser, data structures, tree walking code generator): at least a week's work! Think hard!

- I made the tool! After a fortnight's work, the result was `datadec` - in the `06.datadec` directory (also installed throughout DoC labs). After installing it, use as follows:
- In `07.datadec-eg` you'll find an input file `types.in` containing:


```
TYPE {
    intlist = nil or cons( int head, intlist tail );
    tree    = leaf( string name )
              or node( tree left, tree right );
}
```
- To generate a C module called `datatypes` from `types.in`, invoke:


```
datadec datatypes types.in
```
- This creates `datatypes.c` and `datatypes.h`, two normal looking C files, you can read them, write test programs against the interface, use them in production code with no license restrictions. But don't modify these files - if you do then you can't...
- ... change `types.in` later - suppose you realise that a tree node also needs to store a name (just as the leaves do). Change the type defn, rerun `datadec`. The `tree_node()` constructor now takes 3 arguments!

- Let's look inside `datatypes.h`, to find what `tree` functions `datadec` generates, and how to use them.
- There are two `constructor functions`, one for each *shape of tree*:


```
extern tree tree_leaf( string name );
extern tree tree_node( tree l, tree r );
```
- So, this allows us to build trees as in:


```
tree t1 = tree_leaf( "absolutely" );
tree t2 = tree_leaf( "fabulous" );
tree t  = tree_node( t1, t2 );
```
- Then a function telling you *which shape a tree is*: is it a leaf or a node?


```
typedef enum { tree_is_leaf, tree_is_node } kind_of_tree;
extern kind_of_tree tree_kind( tree t );
```
- Then two `deconstructor functions` which, given a tree of the appropriate shape, breaks it into its constituent pieces:


```
extern void get_tree_leaf( tree t, string *namep );
extern void get_tree_node( tree t, tree *lp, tree *rp );
```

- These allow you to write `tree-walking` code like this leaf-counter:

```
int nleaves( tree t )
{
    if( tree_kind(t) == tree_is_leaf )
    {
        string name; get_tree_leaf( t, &name );
        return 1;          // leaf( name ): contains 1 leaf.
    } else
    {
        tree l, r; get_tree_node( t, &l, &r );
        // node( l, r ): process l and r trees.
        return nleaves(l) + nleaves(r);
    }
}
```

- In Haskell, this'd be:

```
nleaves(leaf(name)) = 1
nleaves(node(l,r))  = nleaves(l) + nleaves(r)
```

- The final function prints a tree to a writable file handle, in human readable format:


```
extern void print_tree( FILE *out, tree t );
```
- To see all the above in use, see `mintesttree.c`.
- By default, `datadec` does not generate free functions. Why? Hard to do right due to shallow vs deep considerations.
- You can now run `datadec -f.` to get experimental `free_TYPE()` functions, although you still have to be careful using these - see the README file for details.
- Looking back, I now view the `fortnight` I spent building `datadec` (and, more recently, the day or two adding `free_TYPE()` support) as the *single best investment of programming time* in my career. I have saved *hundreds of days* programming time using it - and *so can you!*
- You can read a 3-part article I wrote about how I designed `datadec` here:

<http://www.doc.ic.ac.uk/~dcw/PSD/article8/>

Remember:



(and learn Perl, it's great!)