Building your own C Toolkit: Part 1

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

Dept of Computing,
Imperial College London

29th May 2014

When learning any new language, you go through several stages before you achieve basic competence:

When learning any new language, you go through several stages before you achieve basic competence:

- Learn the syntax.
- Learn the semantics.
- Learn the more tricky bits of semantics, eg. pointers (malloc(), free()) and the related issues of shallow vs deep copies.
- Learn the standard library (strcpy(), printf(), qsort(), bsearch()..).
- Learn how to write multi-module programs.
- Learn the idioms and best practices.
- Learn how to write portable code.

When learning any new language, you go through several stages before you achieve basic competence:

- Learn the syntax.
- Learn the semantics.
- Learn the more tricky bits of semantics, eg. pointers (malloc(), free()) and the related issues of shallow vs deep copies.
- Learn the standard library (strcpy(), printf(), qsort(), bsearch()..).
- Learn how to write multi-module programs.
- Learn the idioms and best practices.
- Learn how to write portable code.

These lectures try to answer: What comes after basic C competence?

When learning any new language, you go through several stages before you achieve basic competence:

- Learn the syntax.
- Learn the semantics.
- Learn the more tricky bits of semantics, eg. pointers (malloc(), free()) and the related issues of shallow vs deep copies.
- Learn the standard library (strcpy(), printf(), qsort(), bsearch()..).
- Learn how to write multi-module programs.
- Learn the idioms and best practices.
- Learn how to write portable code.

These lectures try to answer: What comes after basic C competence?

- Craftsmanship!

When learning any new language, you go through several stages before you achieve basic competence:

- Learn the syntax.
- Learn the semantics.
- Learn the more tricky bits of semantics, eg. pointers (malloc(), free()) and the related issues of shallow vs deep copies.
- Learn the standard library (strcpy(), printf(), qsort(), bsearch()..).
- Learn how to write multi-module programs.
- Learn the idioms and best practices.
- Learn how to write portable code.

These lectures try to answer: What comes after basic C competence?

- Craftsmanship!
- Build your own toolkit of useful tools and craft skills to make C programming easier and more productive.

When learning any new language, you go through several stages before you achieve basic competence:

- Learn the syntax.
- Learn the semantics.
- Learn the more tricky bits of semantics, eg. pointers (malloc(), free()) and the related issues of shallow vs deep copies.
- Learn the standard library (strcpy(), printf(), qsort(), bsearch()..).
- Learn how to write multi-module programs.
- Learn the idioms and best practices.
- Learn how to write portable code.

These lectures try to answer: What comes after basic C competence?

- Craftsmanship!
- Build your own toolkit of useful tools and craft skills to make C programming easier and more productive.
- Occasionally: build your own tools!

When learning any new language, you go through several stages before you achieve basic competence:

- Learn the syntax.
- Learn the semantics.
- Learn the more tricky bits of semantics, eg. pointers (malloc(), free()) and the related issues of shallow vs deep copies.
- Learn the standard library (strcpy(), printf(), qsort(), bsearch()..).
- Learn how to write multi-module programs.
- Learn the idioms and best practices.
- Learn how to write portable code.

These lectures try to answer: What comes after basic C competence?

- Craftsmanship!
- Build your own toolkit of useful tools and craft skills to make C programming easier and more productive.
- Occasionally: build your own tools!
- Principle: ruthless automation - when doing something boring and repetitive, think: can I save time by automating this?

Today, and the next two Thursdays, I'll show you some of the tools in my toolkit, in the hope they'll be useful to you! Today, we'll cover:

- Programmer's Editors: Use a single editor well.
- Automating Compilation (reminder): Use make.
- Automating Testing: Test often, test ruthlessly.
- Debugging: Use a debugger and know it well.
- Building shortlived tools on the fly.

Today, and the next two Thursdays, I'll show you some of the tools in my toolkit, in the hope they'll be useful to you! Today, we'll cover:

- Programmer's Editors: Use a single editor well.
- Automating Compilation (reminder): Use make.
- Automating Testing: Test often, test ruthlessly.
- Debugging: Use a debugger and know it well.
- Building shortlived tools on the fly.

Notes:

- I strongly recommend The Pragmatic Programmer (PP) book, by Hunt & Thomas. The woodworking metaphor - and a series of excellent programming Tips - comes from there.

Today, and the next two Thursdays, I'll show you some of the tools in my toolkit, in the hope they'll be useful to you! Today, we'll cover:

- Programmer's Editors: Use a single editor well.
- Automating Compilation (reminder): Use make.
- Automating Testing: Test often, test ruthlessly.
- Debugging: Use a debugger and know it well.
- Building shortlived tools on the fly.

Notes:

- I strongly recommend The Pragmatic Programmer (PP) book, by Hunt & Thomas. The woodworking metaphor - and a series of excellent programming Tips - comes from there.
- There's a tarball of examples associated with each lecture, as a shorthand tarball 01.list refers to the directory called **01.list** inside the tarball. Each directory contains a README file.

- Hunt & Thomas write (in Tip 22):

  > *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.

- Hunt & Thomas write (in Tip 22):

  > *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.

- Hunt & Thomas write (in Tip 22):

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

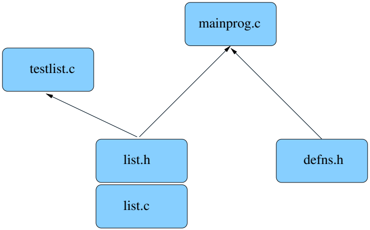
- Not my business to tell you which editor to use; avoid editor wars.
- IDEs such as Eclipse provide an editor, an automated compilation system and a debugging environment. If you're going to use an IDE, invest time learning how to use it well, and how to extend and program it.
- I use Vi/Vim, terse but powerful, extensible in several ways - eg. macros and a "pipe through external command" mechanism.

- Hunt & Thomas write (in Tip 22):

  > *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/Vim, 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.
- After initial exploration of the possibilities, learn your chosen editor thoroughly and become expert in its use, including how to plug external tools into it.

When multi-file C programming, eg:



Many source 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 impln vs interface).
- testlist.c includes list.h
- mainprog.c includes list.h and defns.h

Make uses such file dependencies, encoded in a **Makefile**, to automatically compile your programs. A Makefile contains dependency rules between target and source files with actions (commands) to generate each target from its' sources.

Here's the Makefile for the multi-module example:

Here's the Makefile for the multi-module example:

```
CC      = gcc
CFLAGS  = -Wall
PROGS   = testlist mainprog
```

Here's the Makefile for the multi-module example:

```
CC      = gcc
CFLAGS  = -Wall
PROGS   = testlist mainprog

all:    $(PROGS)
clean:
        /bin/rm -f $(PROGS) *.o core
```

Here's the Makefile for the multi-module example:

```
CC      = gcc
CFLAGS  = -Wall
PROGS   = testlist mainprog

all:    $(PROGS)
clean:
        /bin/rm -f $(PROGS) *.o core

testlist:       testlist.o list.o
mainprog:       mainprog.o list.o
mainprog.o:     list.h defns.h
testlist.o:     list.h
list.o:         list.h
```

Here's the Makefile for the multi-module example:

```
CC      = gcc
CFLAGS  = -Wall
PROGS   = testlist mainprog

all:    $(PROGS)
clean:
        /bin/rm -f $(PROGS) *.o core

testlist:       testlist.o list.o
mainprog:       mainprog.o list.o
mainprog.o:     list.h defns.h
testlist.o:     list.h
list.o:         list.h
```

- If list.h is altered, then list.c, testlist.c and mainprog.c need recompiling, and testlist and mainprog need relinking against the list object file (list.o).

Here's the Makefile for the multi-module example:

```
CC      = gcc
CFLAGS  = -Wall
PROGS   = testlist mainprog

all:    $(PROGS)
clean:
        /bin/rm -f $(PROGS) *.o core

testlist:       testlist.o list.o
mainprog:       mainprog.o list.o
mainprog.o:     list.h defns.h
testlist.o:     list.h
list.o:         list.h
```

- If list.h is altered, then list.c, testlist.c and mainprog.c need recompiling, and testlist and mainprog need relinking against the list object file (list.o).
- Summary: Always use make. Keep your Makefile up to date.
- Exercise: why not auto generate your Makefiles? Many tools generate Makefiles automatically, easy to write.

- In Tip 62, 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!

- In Tip 62, 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.

- In Tip 62, 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.

- 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:

- 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:
- First, recompile all source code with gcc -g flag:

- 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:
- First, recompile all source code with gcc -g flag:
  - Set `CFLAGS = -Wall -g` in your Makefile.

- 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:
- First, recompile all source code with gcc -g flag:
  - Set `CFLAGS = -Wall -g` in your Makefile.
  - Recompile everything via `make clean all`.

- 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:
- First, recompile all source code with gcc -g flag:
  - Set `CFLAGS = -Wall -g` in your Makefile.
  - 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.

- 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:
- First, recompile all source code with gcc -g flag:
  - Set `CFLAGS = -Wall -g` in your Makefile.
  - 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.
- Also, `up` and `down` move up or down one level on the frame stack.

- `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.

- `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.
  - `x/12xb &str` as hexadecimal etc.

- `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.
  - `x/12xb &str` as hexadecimal etc.
  - `help x` (inside gdb) for more info.

- `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.
  - `x/12xb &str` 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`.

- We often find ourselves writing hundreds of repetitive "pattern instances", 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); }
...
```

- We often find ourselves writing hundreds of repetitive “pattern instances”, 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); }
...
```

- Generate such lines automatically using a shortlived tool, scaffolding that you build on demand, use a few times, then discard: All that varies from line to line is (funcname,operator), eg. `(plus,+)`.

- We often find ourselves writing hundreds of repetitive "pattern instances", 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); }
...
```

- Generate such lines automatically using a shortlived tool, scaffolding that you build on demand, use a few times, then discard: All that varies from line to line 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); }"
```

- We often find ourselves writing hundreds of repetitive "pattern instances", 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); }
...
```

- Generate such lines automatically using a shortlived tool, scaffolding that you build on demand, use a few times, then discard: All that varies from line to line 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 about two minutes:

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

- We often find ourselves writing hundreds of repetitive “pattern instances”, 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); }
...
```

- Generate such lines automatically using a shortlived tool, scaffolding that you build on demand, use a few times, then discard: All that varies from line to line 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 about two minutes:

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

- Don’t know Perl? write it in C instead - took me 15 minutes using standard library function strtok(). See 03.tiny-tool/genfuncs1.c for a C implementation.

- We often find ourselves writing hundreds of repetitive "pattern instances", 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); }
...
```

- Generate such lines automatically using a shortlived tool, scaffolding that you build on demand, use a few times, then discard: All that varies from line to line 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 about two minutes:

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

- Don't know Perl? write it in C instead - took me 15 minutes using standard library function strtok(). See 03.tiny-tool/genfuncs1.c for a C implementation.
- 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 when your needs change, or just for your convenience:
- 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' < i
```

- Prefix the typename onto function names, eg. `int_plus`:

```
perl -nle '($f,$op)=split(/,/); printf "int %-15s( int a, int b ) { return (a${op}b); }\n", "int_${
```

- Noticing all those "int"s, let's make it easier to change:

```
perl -nle '$t="int"; ($f,$op)=split(/,/);
           printf "${t} %-15s( ${t} a, $t b ) { return (a${op}b); }\n", "${t}_${f}"' < input
```

- We could let the user set the type within the input, perhaps the first line of input, see 03.tiny-tool/README for details.
- More usefully, 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
       special case: if F=="TYPE" then T=Op
OUTPUT:
  foreach F, Op pair where F!="TYPE":
   "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 "${t} %-15s( ${t} a, ${t} b ) { return (a${op}b); }\n", "${t}_${f}"' < input
```

- See 03.tiny-tool/genfuncs3.c for a C implementation.
- Final thought, instead of hardcoding the output format in the printf, we could replace TYPEs with output TEMPLATEs, 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:

```
perl -nle '@f=split(/,/,$_,2); if( $f[0] eq "TEMPLATE" ) { $t=$f[1]; next; }
           $_=$t; s/<(\d+)>/$f[$1]/g; print' < input
```

- This is now a very simple macro processor.