# Building your own C Toolkit: Part 1

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

Dept of Computing,
Imperial College London

30th May 2013

- When learning any new language, you go through several stages:

- When learning any new language, you go through several stages:
  - Getting familiar with the basic syntax.
  - Getting familiar with the basic semantics.
  - Getting familiar with the more tricky bits of semantics.
  - Getting familiar with pointers (malloc() etc).
  - Getting familiar with the standard library (strcpy(), printf(), qsort() etc).
  - Getting familiar with writing multi-module programs.

- When learning any new language, you go through several stages:
  - Getting familiar with the basic syntax.
  - Getting familiar with the basic semantics.
  - Getting familiar with the more tricky bits of semantics.
  - Getting familiar with pointers (malloc() etc).
  - Getting familiar with the standard library (strcpy(), printf(), qsort() etc).
  - Getting familiar with writing multi-module programs.
- At this point, I suggest that you're competent in C. What do you do next?

- When learning any new language, you go through several stages:
  - Getting familiar with the basic syntax.
  - Getting familiar with the basic semantics.
  - Getting familiar with the more tricky bits of semantics.
  - Getting familiar with pointers (malloc() etc).
  - Getting familiar with the standard library (strcpy(), printf(), qsort() etc).
  - Getting familiar with writing multi-module programs.
- At this point, I suggest that you're competent in C. What do you do next?
- Like a carpenter, build your own toolkit of useful tools to make C programming easier and more productive.

- When learning any new language, you go through several stages:
  - Getting familiar with the basic syntax.
  - Getting familiar with the basic semantics.
  - Getting familiar with the more tricky bits of semantics.
  - Getting familiar with pointers (malloc() etc).
  - Getting familiar with the standard library (strcpy(), printf(), qsort() etc).
  - Getting familiar with writing multi-module programs.
- At this point, I suggest that you're competent in C. What do you do next?
- 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!

- When learning any new language, you go through several stages:
  - Getting familiar with the basic syntax.
  - Getting familiar with the basic semantics.
  - Getting familiar with the more tricky bits of semantics.
  - Getting familiar with pointers (malloc() etc).
  - Getting familiar with the standard library (strcpy(), printf(), qsort() etc).
  - Getting familiar with writing multi-module programs.
- At this point, I suggest that you're competent in C. What do you do next?
- 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, and next Thursday, I'd like to show you some of the tools in my toolkit, hopefully they'll be useful to you! Today, 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.

Today, and next Thursday, I'd like to show you some of the tools in my toolkit, hopefully they'll be useful to you! Today, 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.

Today, and next Thursday, I'd like to show you some of the tools in my toolkit, hopefully they'll be useful to you! Today, 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.
- 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 next Thursday, I'd like to show you some of the tools in my toolkit, hopefully they'll be useful to you! Today, 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.
- 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 describing what's in it - in great detail.

- 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.*

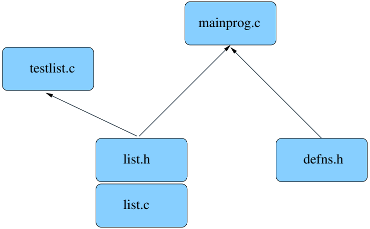
- Not my business to tell you which editor to use; avoid editor wars.
- IDEs such as Eclipse provide an editor, an automated compilation system and a debugging environment. If you're going to use an IDE, invest time learning how to use it well, and how to extend and program it.
- I use vi, terse but 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, 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 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.

- `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`.

- Irritating C problem: keeping the prototype declarations in interfaces (.h files) in sync with the function definitions in the implementation (.c files).

- 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.

- 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.

- 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!

- 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.

- 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.

- 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.

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.

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)!

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.

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.
- derefencing an uninitialized/reclaimed pointer gives non-deterministic behaviour (really hard to debug!).

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.
- 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?

- 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:

- 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.

- 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 install libmem from tarball directory 04.libmem

- 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 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.

- 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 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

- 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 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! They tell you exactly what you've done wrong! Magic!

- 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 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! They tell you exactly what you've done wrong! 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?

- 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.

- 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:-)

- 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 should be linear with N. Test it with `time ./iterate N 0` for several values of N, graph results.

- 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 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).

- 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!

- 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!

- 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'

- 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.

- 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!

- 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 01.list) runs cleanly with libmem. (Import CFLAGS and LDLIBS from 05.mem-eg's Makefile).