Building your own C Toolkit: Part 2

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

Dept of Computing,
Imperial College London

30th May 2019

The handout and tarballs are available on CATE and at:

http://www.doc.ic.ac.uk/~dcw/c-tools-2019/lecture2/

- Last lecture, we introduced the idea of building a C programming toolkit. and covered programmer's editors, make and multi-directory programs.

- Last lecture, we introduced the idea of building a C programming toolkit. and covered programmer's editors, make and multi-directory programs.
- Today, we're going to carry on, and cover:
  - What to do when things go wrong.
  - Debugging: gdb.

- Last lecture, we introduced the idea of building a C programming toolkit. and covered programmer's editors, make and multi-directory programs.
- Today, we're going to carry on, and cover:
  - What to do when things go wrong.
  - Debugging: gdb.
  - Detecting memory leaks: valgrind.
  - Profiling-led Optimization.
  - Automatic Ruthless Testing.

- Last lecture, we introduced the idea of building a C programming toolkit. and covered programmer's editors, make and multi-directory programs.
- Today, we're going to carry on, and cover:
    - What to do when things go wrong.
    - Debugging: gdb.
    - Detecting memory leaks: valgrind.
    - Profiling-led Optimization.
    - Automatic Ruthless Testing.
- But first: one thing I didn't make clear last week was the destination of all our C Tools: the `~/c-tools` directory. This was explained in the first tarball's README.

- Last lecture, we introduced the idea of building a C programming toolkit. and covered programmer's editors, make and multi-directory programs.
- Today, we're going to carry on, and cover:
  - What to do when things go wrong.
  - Debugging: gdb.
  - Detecting memory leaks: valgrind.
  - Profiling-led Optimization.
  - Automatic Ruthless Testing.
- But first: one thing I didn't make clear last week was the destination of all our C Tools: the `~/c-tools` directory. This was explained in the first tarball's README.
- One of the most common things that you will experience with C programming is that your program dies mysteriously with a Segmentation Fault (aka a segfault).
- Why is that?

- Last lecture, we introduced the idea of building a C programming toolkit. and covered programmer's editors, make and multi-directory programs.
- Today, we're going to carry on, and cover:
  - What to do when things go wrong.
  - Debugging: gdb.
  - Detecting memory leaks: valgrind.
  - Profiling-led Optimization.
  - Automatic Ruthless Testing.
- But first: one thing I didn't make clear last week was the destination of all our C Tools: the `~/c-tools` directory. This was explained in the first tarball's README.
- One of the most common things that you will experience with C programming is that your program dies mysteriously with a Segmentation Fault (aka a segfault).
- Why is that?
- Because C assumes you know what you're doing!

- Or..


ATTENTION
Your mother doesn't work here.
Please clean up your own mess!

- Or..

- It's your responsibility to: check that you don't overrun the bounds of an array,

- Or..

- It's your responsibility to: check that you don't overrun the bounds of an array, don't dereference a NULL/bad pointer,

- Or..

- It's your responsibility to: check that you don't overrun the bounds of an array, don't dereference a NULL/bad pointer, and don't write into read-only memory - as in `char *p = "get ready"; *p = 's';` or `strcpy(p,"hello");`

- Our first technique for fixing a broken C program - when it crashes or produces the wrong answers - is to debug it.
- As the Pragmatic Programmers so nearly said: Know a single debugger well.

- Our first technique for fixing a broken C program - when it crashes or produces the wrong answers - is to debug it.
- As the Pragmatic Programmers so nearly said: Know a single debugger well.
- Let's use gdb, the GNU debugger, to understand a problem in `01.string-debug` - a program crashing with a segfault.

- Our first technique for fixing a broken C program - when it crashes or produces the wrong answers - is to debug it.
- As the Pragmatic Programmers so nearly said: Know a single debugger well.
- Let's use gdb, the GNU debugger, to understand a problem in `01.string-debug` - a program crashing with a segfault.
- The `README` in 01.string-debug explains what to do. In summary:
- Recompile all source code with debugging support - add gcc's -g flag to `CFLAGS` in the Makefile and type `make clean all`.

- Our first technique for fixing a broken C program - when it crashes or produces the wrong answers - is to debug it.
- As the Pragmatic Programmers so nearly said: Know a single debugger well.
- Let's use gdb, the GNU debugger, to understand a problem in `01.string-debug` - a program crashing with a segfault.
- The `README` in 01.string-debug explains what to do. In summary:
- Recompile all source code with debugging support - add gcc's -g flag to `CFLAGS` in the Makefile and type `make clean all`.
- Start gdb then run the program, interacting with it until it crashes.
- Now type `where` to see the call frame stack - the sequence of function calls leading to the crash.

- Our first technique for fixing a broken C program - when it crashes or produces the wrong answers - is to debug it.
- As the Pragmatic Programmers so nearly said: Know a single debugger well.
- Let's use gdb, the GNU debugger, to understand a problem in `01.string-debug` - a program crashing with a segfault.
- The `README` in 01.string-debug explains what to do. In summary:
- Recompile all source code with debugging support - add gcc's -g flag to `CFLAGS` in the Makefile and type `make clean all`.
- Start gdb then run the program, interacting with it until it crashes.
- Now type `where` to see the call frame stack - the sequence of function calls leading to the crash.
- Then print out the values and types of variables to see what has gone wrong.
- The `p VARIABLE` command prints out a variable, and the `whatis VARIABLE` command reminds you of it's type.

- In particular, you'll see that the char * variable q has a corrupt pointer in it: `p q` shows the error: `Cannot access memory at address 0x555500657265`
- By printing the addresses of variables `p, q` and `str` (by commands like `p &str` etc) we can see that q happens to follow `str` in memory.
- We can then use gdb's memory dumper to show us the chunk of memory starting at `&str`, using the `x/12c &str` command:

```
0x555555755020 <str>:   104 'h' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' '  116 't'104 'h'
0x555555755028 <q>:     101 'e' 114 'r' 101 'e' 0 '\000'
```

- Do you see the problem now?

- In particular, you'll see that the char * variable q has a corrupt pointer in it: `p q` shows the error: `Cannot access memory at address 0x555500657265`
- By printing the addresses of variables `p, q` and `str` (by commands like `p &str` etc) we can see that q happens to follow `str` in memory.
- We can then use gdb's memory dumper to show us the chunk of memory starting at `&str`, using the `x/12c &str` command:

```
0x555555755020 <str>:   104 'h' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' '  116 't'104 'h'
0x555555755028 <q>:     101 'e' 114 'r' 101 'e' 0 '\000'
```

- Do you see the problem now? str is a char [8] but we have copied `"hello there"` into it - more than 8 chars,

- In particular, you'll see that the char * variable q has a corrupt pointer in it: `p q` shows the error: `Cannot access memory at address 0x555500657265`
- By printing the addresses of variables `p, q` and `str` (by commands like `p &str` etc) we can see that q happens to follow `str` in memory.
- We can then use gdb's memory dumper to show us the chunk of memory starting at `&str`, using the `x/12c &str` command:

```
0x555555755020 <str>:   104 'h' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' '  116 't'104 'h'
0x555555755028 <q>:     101 'e' 114 'r' 101 'e' 0 '\000'
```

- Do you see the problem now? str is a char [8] but we have copied `"hello there"` into it - more than 8 chars, so the rest of the string ('e', 'r', 'e' and the trailing \0) has OVERFLOWED into the adjacent variable's space, which happens to be q.

- In particular, you'll see that the char * variable q has a corrupt pointer in it: `p q` shows the error: `Cannot access memory at address 0x555500657265`
- By printing the addresses of variables `p`, `q` and `str` (by commands like `p &str` etc) we can see that q happens to follow `str` in memory.
- We can then use gdb's memory dumper to show us the chunk of memory starting at `&str`, using the `x/12c &str` command:

```
0x555555755020 <str>:   104 'h' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' '  116 't'104 'h'
0x555555755028 <q>:     101 'e' 114 'r' 101 'e' 0 '\000'
```

- Do you see the problem now? str is a char [8] but we have copied `"hello there"` into it - more than 8 chars, so the rest of the string ('e', 'r', 'e' and the trailing \0) has OVERFLOWED into the adjacent variable's space, which happens to be q.
- But q is a char *, so interpreting those overflowing bytes as an address we get 0x555500657265, some arbitrary address in memory. Fortunately, that's not a valid char *, so dereferencing it gave our segfault.

- As to finding out how the overflow occurred (if it's not obvious), you can use gdb to set breakpoints, or watch a variable to stop the debugging session each time it changes.
- Using the `watch q` command, and then running the program, we find that q was modified accidentally inside `append()`, specifically where we strcat() without checking that the concatenated string fits.
- The README file suggests an obvious two-part fix for the problem:
    - First, write additional code inside `append()` to detect overflow, and use `assert()` to blow up the program when overflow does occur.
    - Second, prevent overflow from occurring this time - by making `char str[8]` bigger!
- Google for gdb tutorial for more info.
- Most important, leave gdb by `quit`.

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 try to 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 try to 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 try to use more memory than the computer has physical RAM!
- free()ing a block twice is equally dangerous.
- Dereferencing an uninitialized/reclaimed pointer gives Undefined 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 try to use more memory than the computer has physical RAM!
- free()ing a block twice is equally dangerous.
- Dereferencing an uninitialized/reclaimed pointer gives Undefined Behaviour (really hard to debug!).
- Even when you get Seg faults - gdb where (frame stack) may show it crashes in system libraries - but it doesn't really!

To diagnose such problems, we use tools like valgrind:

- Suppose we have a pre-written, pre-tested hash table module, plus a unit test program **testhash**.
- It passes all tests (creating a single hash table, populating it with keys and values, finding keys, iterating over the hash, then freeing the hash table).
- We've even used it in several successful projects - so we're confident that it works!
- But we have never checked for memory leaks with valgrind!

- Suppose we have a pre-written, pre-tested hash table module, plus a unit test program **testhash**.
- It passes all tests (creating a single hash table, populating it with keys and values, finding keys, iterating over the hash, then freeing the hash table).
- We've even used it in several successful projects - so we're confident that it works!
- But we have never checked for memory leaks with valgrind!
- 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**.
- It passes all tests (creating a single hash table, populating it with keys and values, finding keys, iterating over the hash, then freeing the hash table).
- We've even used it in several successful projects - so we're confident that it works!
- But we have never checked for memory leaks with valgrind!
- 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**.
- It passes all tests (creating a single hash table, populating it with keys and values, finding keys, iterating over the hash, then freeing the hash table).
- We've even used it in several successful projects - so we're confident that it works!
- But we have never checked for memory leaks with valgrind!
- 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:-)
- Write a new test program `iterate N M` that (silently) performs all previous tests N times, sleeping M seconds afterwards.

- Behaviour of `iterate` (with M=0) should be linear with N. Test it with `time ./iterate N 0` for several values of N.

- Behaviour of `iterate` (with M=0) should be linear with N. Test it with `time ./iterate N 0` for several values of N.
- However, we find some strange behaviour around 30k iterations on a 16GB lab machine (eg point07): Goes slower than it should, CPU %age falls, the process starts doing I/O and page faults.
- What on earth is happening?
- Try monitoring with `top`, sorting by %age of memory used (either within top via the 'f' command, or via a shell alias: `alias memtop='top -o %MEM'`).

- Behaviour of `iterate` (with M=0) should be linear with N. Test it with `time ./iterate N 0` for several values of N.
- However, we find some strange behaviour around 30k iterations on a 16GB lab machine (eg point07): Goes slower than it should, CPU %age falls, the process starts doing I/O and page faults.
- What on earth is happening?
- Try monitoring with `top`, sorting by %age of memory used (either within top via the 'f' command, or via a shell alias: `alias memtop='top -o %MEM'`).
- Run iterate with a time delay: `time ./iterate 36000 10` and watch top! iterate's memory use grows and grows, eventually hits 95% of physical memory. At this point the system starts swapping (%wait goes busy), and the load average increases.

- Behaviour of `iterate` (with M=0) should be linear with N. Test it with `time ./iterate N 0` for several values of N.
- However, we find some strange behaviour around 30k iterations on a 16GB lab machine (eg point07): Goes slower than it should, CPU %age falls, the process starts doing I/O and page faults.
- What on earth is happening?
- Try monitoring with `top`, sorting by %age of memory used (either within top via the 'f' command, or via a shell alias: `alias memtop='top -o %MEM'`).
- Run iterate with a time delay: `time ./iterate 36000 10` and watch top! iterate's memory use grows and grows, eventually hits 95% of physical memory. At this point the system starts swapping (%wait goes busy), and the load average increases.
- Hypothesis: the hash table module is leaking some memory!

- Behaviour of `iterate` (with M=0) should be linear with N. Test it with `time ./iterate N 0` for several values of N.
- However, we find some strange behaviour around 30k iterations on a 16GB lab machine (eg point07): Goes slower than it should, CPU %age falls, the process starts doing I/O and page faults.
- What on earth is happening?
- Try monitoring with `top`, sorting by %age of memory used (either within top via the 'f' command, or via a shell alias: `alias memtop='top -o %MEM'`).
- Run iterate with a time delay: `time ./iterate 36000 10` and watch top! iterate's memory use grows and grows, eventually hits 95% of physical memory. At this point the system starts swapping (%wait goes busy), and the load average increases.
- Hypothesis: the hash table module is leaking some memory! This is a job for valgrind!

- Run `valgrind ./testhash` [a simpler test program]
- The result is:

```
LEAK SUMMARY:
   definitely lost: 520,528 bytes in 2 blocks
...
Rerun with --leak-check=full to see details..
```

- Run `valgrind ./testhash` [a simpler test program]
- The result is:

```
LEAK SUMMARY:
   definitely lost: 520,528 bytes in 2 blocks
...
Rerun with --leak-check=full to see details..
```

- Run `valgrind --leak-check=full ./testhash` and you see:

```
260,264 bytes in 1 blocks are definitely lost in loss record 1 of 2
    at 0x4C2FB0F: malloc..
    by 0x108F65: hashCreate (hash.c:73)
    by 0x108C69: main (testhash.c:91)

260,264 bytes in 1 blocks are definitely lost in loss record 2 of 2
    at 0x4C2FB0F: malloc..
    by 0x109059: hashCopy (hash.c:112)
    by 0x108E4C: main (testhash.c:123)
```

- Look at line 73 of `hash.c` in `hashCreate()`, it reads:

```
h->data = (tree *) malloc( NHASH*sizeof(tree) );
```

- and line 112 is nearly identical in `hashCopy()`.
- Where might we expect to free this "hash data"? Look in the corresponding `hashFree(hash h)` function.

- Look at line 73 of `hash.c` in `hashCreate()`, it reads:

```
h->data = (tree *) malloc( NHASH*sizeof(tree) );
```

- and line 112 is nearly identical in `hashCopy()`.
- Where might we expect to free this "hash data"? Look in the corresponding `hashFree(hash h)` function.
- Aha! `h->data` is NOT FREED. A simple mistake!
- Add the missing `free(h->data)`, recompile (make).

- Look at line 73 of `hash.c` in `hashCreate()`, it reads:

  ```
  h->data = (tree *) malloc( NHASH*sizeof(tree) );
  ```

- and line 112 is nearly identical in `hashCopy()`.
- Where might we expect to free this "hash data"? Look in the corresponding `hashFree(hash h)` function.
- Aha! `h->data` is NOT FREED. A simple mistake!
- Add the missing `free(h->data)`, recompile (make).
- Rerun `valgrind ./testhash` and it reports no leaking blocks.
- Run `./iterate 36000` again - no non linear behaviour, no weird slowdown.

- Look at line 73 of `hash.c` in `hashCreate()`, it reads:

```
h->data = (tree *) malloc( NHASH*sizeof(tree) );
```

- and line 112 is nearly identical in `hashCopy()`.
- Where might we expect to free this "hash data"? Look in the corresponding `hashFree(hash h)` function.
- Aha! `h->data` is NOT FREED. A simple mistake!
- Add the missing `free(h->data)`, recompile (make).
- Rerun `valgrind ./testhash` and it reports no leaking blocks.
- Run `./iterate 36000` again - no non linear behaviour, no weird slowdown.
- Summary: use valgrind regularly while developing your code. Save yourself loads of grief, double your confidence.
- Exercise: does the list example (in Lecture 1's 01.intlist - or any of the later versions) run cleanly with valgrind?

- gcc and most other C compilers can be asked to optimize the code they generate, gcc's options for this are -O, -O2, -O3. Worth trying, rarely makes a significant difference.

- gcc and most other C compilers can be asked to optimize the code they generate, gcc's options for this are -O, -O2, -O3. Worth trying, rarely makes a significant difference.
- What makes far more difference is finding the hot spots using a profiler and selectively optimizing them - by hand. This can produce dramatic speedups, and profiling often (always?) produces surprises.

- gcc and most other C compilers can be asked to optimize the code they generate, gcc's options for this are -O, -O2, -O3. Worth trying, rarely makes a significant difference.
- What makes far more difference is finding the hot spots using a profiler and selectively optimizing them - by hand. This can produce dramatic speedups, and profiling often (always?) produces surprises.
- Let's profile the bugfixed hash module's iterate test program (in the `03.hash-profile` directory):
  - 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 `make profile` to generate a profile, this does 2 things:

- gcc and most other C compilers can be asked to optimize the code they generate, gcc's options for this are -O, -O2, -O3. Worth trying, rarely makes a significant difference.
- What makes far more difference is finding the hot spots using a profiler and selectively optimizing them - by hand. This can produce dramatic speedups, and profiling often (always?) produces surprises.
- Let's profile the bugfixed hash module's iterate test program (in the `03.hash-profile` directory):
  - 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 `make profile` to generate a profile, this does 2 things:
  - First, it runs `./iterate 10000`, which runs slower than normal while profiling, and produces a binary profiling file called gmon.out.
  - Second, 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.

- head profile.orig shows results like:

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  us/call  us/call  name
25.64      0.73     0.73 650660000     0.00     0.00  free_tree
22.83      1.38     0.65    20000    32.53    69.07  hashFree
21.43      1.99     0.61    10000    61.06    61.06  hashCreate
15.45      2.43     0.44    10000    44.04    86.09  hashCopy
14.75      2.85     0.42 325330000     0.00     0.00  copy_tree
```

- head profile.orig shows results like:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 25.64      0.73     0.73 650660000     0.00     0.00  free_tree
 22.83      1.38     0.65    20000    32.53    69.07  hashFree
 21.43      1.99     0.61    10000    61.06    61.06  hashCreate
 15.45      2.43     0.44    10000    44.04    86.09  hashCopy
 14.75      2.85     0.42 325330000     0.00     0.00  copy_tree
```

- 650 million calls to free_tree() and 325 million calls to copy_tree() are suspicious. First, 65066 is twice 32533! Aha! the hash table's dynamic array of binary 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. Then profile again, a new hotspot may appear.

- head profile.orig shows results like:

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  us/call  us/call  name
25.64      0.73     0.73 650660000     0.00     0.00  free_tree
22.83      1.38     0.65    20000    32.53    69.07  hashFree
21.43      1.99     0.61    10000    61.06    61.06  hashCreate
15.45      2.43     0.44    10000    44.04    86.09  hashCopy
14.75      2.85     0.42 325330000     0.00     0.00  copy_tree
```

- 650 million calls to free_tree() and 325 million calls to copy_tree() are suspicious. First, 65066 is twice 32533! Aha! the hash table's dynamic array of binary 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. Then profile again, a new hotspot may appear.
- We might also consider shrinking the size of the array to some smaller prime number.

- head profile.orig shows results like:

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  us/call  us/call  name
25.64      0.73     0.73 650660000     0.00     0.00  free_tree
22.83      1.38     0.65    20000    32.53    69.07  hashFree
21.43      1.99     0.61    10000    61.06    61.06  hashCreate
15.45      2.43     0.44    10000    44.04    86.09  hashCopy
14.75      2.85     0.42 325330000     0.00     0.00  copy_tree
```

- 650 million calls to free_tree() and 325 million calls to copy_tree() are suspicious. First, 65066 is twice 32533! Aha! the hash table's dynamic array of binary 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. Then profile again, a new hotspot may appear.
- We might also consider shrinking the size of the array to some smaller prime number. More radically, dynamically resize the array (and rehash all the keys) when the hash gets too full.

- Pragmatic Programmers Tip 62:

  Test Early, Test Often, Test Automatically:
  Tests that run with every build are much more effective than test plans that sit on a shelf.

- Pragmatic Programmers Tip 62:

  > *Test Early, Test Often, Test Automatically:*
  > *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 integration tests which test a set of modules together, and overall program tests.
- Add make test target to run the tests. Run them frequently.
- Can run make test whenever you commit a new version into git!

- Pragmatic Programmers Tip 62:

  > *Test Early, Test Often, Test Automatically:*
  > *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 integration tests which test a set of modules together, and overall program tests.
- Add make test target to run the tests. Run them frequently.
- Can run make test whenever you commit a new version into git!
- **Most important**: Test programs should check for correct results themselves (essentially, hardcoding the correct answers in them).

- Pragmatic Programmers Tip 62:

  > *Test Early, Test Often, Test Automatically:*
  > *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 integration tests which test a set of modules together, and overall program tests.
- Add make test target to run the tests. Run them frequently.
- Can run make test whenever you commit a new version into git!
- **Most important**: Test programs should check for correct results themselves (essentially, hardcoding the correct answers in them).
- If your "test program" simply prints lots of messages out and relies on a human being to read the output, it's **not a proper test program**.

- Pragmatic Programmers Tip 62:

  > *Test Early, Test Often, Test Automatically:*
  > *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 integration tests which test a set of modules together, and overall program tests.
- Add make test target to run the tests. Run them frequently.
- Can run make test whenever you commit a new version into git!
- **Most important**: Test programs should check for correct results themselves (essentially, hardcoding the correct answers in them).
- If your "test program" simply prints lots of messages out and relies on a human being to read the output, it's **not a proper test program**.
- Helpful if all tests report in a common style. C doesn't come with a testing infrastructure like Java's jUnit, but it's pretty easy to whip something simple up.

- For example:

```
void testbool( bool ok, char *testname )
{
        printf( "T %s: %s\n", testname, ok?"OK":"FAIL" );
}
```

- `testbool()` can be used via:

```
intlist l = intlist_nil();
testbool( intlist_kind( l ) == intlist_is_nil,
     "kind(nil) is nil" );

l = intlist_cons( 100, l );
testbool( intlist_kind( l ) == intlist_is_cons,
     "kind([100]) is cons" );
```

- This produces output of the form:

```
T kind(nil) is nil: OK
T kind([100]) is cons: OK
```

- make test could run all test programs in sequence:

```
test:   testprogram1 testprogram2 ...
        ./testprogram1
        ./testprogram2
```

- Or, given the above fixed test output format, we could post-process the output in the `make test` rule:

```
        ./testprogram1 | grep ^T
        ./testprogram2 | grep ^T
```

- Or we could invoke a simple test framework script with testprograms as arguments, which runs the programs and post-processes the results. eg:

```
test:   testprogram1 testprogram2 ...
        summarisetests ./testprogram1 ./testprogram2
```

- You'll find such a summarisetests Perl script, and `testbool()` in it's own testutils module in the `04.testutils` directory. Go in there and type `make install`, then enter `05.intlist-with-testing` to see intlist with testing.

Test Driven Development (TDD) writes the test programs before implementing the feature to test.

Test Driven Development (TDD) writes the test programs before implementing the feature to test.

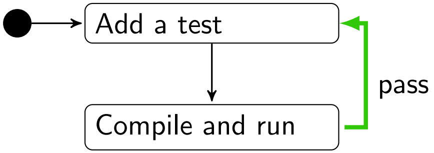
- This helps you focus on one task at a time.
- Encourages incremental development.
- Reduces debugger use.
- (When you find - and fix - a new bug, write a test for it!)
- Don't forget to add some overall tests too.

Test Driven Development (TDD) writes the test programs before implementing the feature to test.

- This helps you focus on one task at a time.
- Encourages incremental development.
- Reduces debugger use.
- (When you find - and fix - a new bug, write a test for it!)
- Don't forget to add some overall tests too.

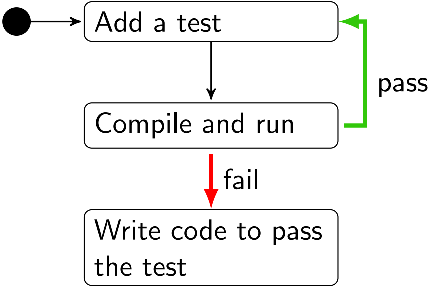
Test Driven Development (TDD) writes the test programs before implementing the feature to test.

- This helps you focus on one task at a time.
- Encourages incremental development.
- Reduces debugger use.
- (When you find - and fix - a new bug, write a test for it!)
- Don't forget to add some overall tests too.



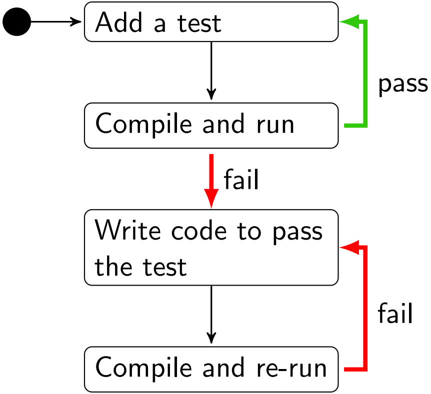
Test Driven Development (TDD) writes the test programs before implementing the feature to test.

- This helps you focus on one task at a time.
- Encourages incremental development.
- Reduces debugger use.
- (When you find - and fix - a new bug, write a test for it!)
- Don't forget to add some overall tests too.



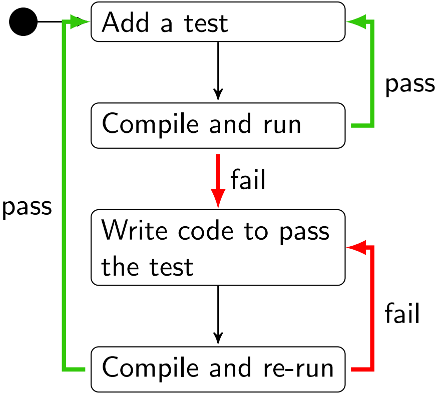
Test Driven Development (TDD) writes the test programs before implementing the feature to test.

- This helps you focus on one task at a time.
- Encourages incremental development.
- Reduces debugger use.
- (When you find - and fix - a new bug, write a test for it!)
- Don't forget to add some overall tests too.



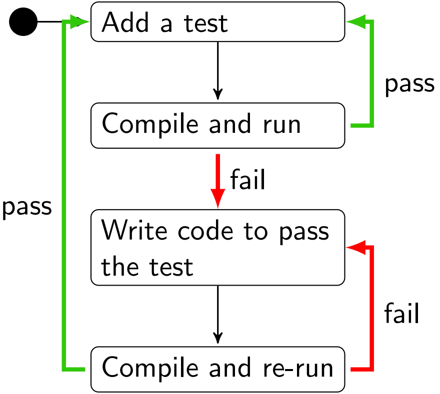
Test Driven Development (TDD) writes the test programs before implementing the feature to test.

- This helps you focus on one task at a time.
- Encourages incremental development.
- Reduces debugger use.
- (When you find - and fix - a new bug, write a test for it!)
- Don't forget to add some overall tests too.



Test Driven Development (TDD) writes the test programs before implementing the feature to test.

- This helps you focus on one task at a time.
- Encourages incremental development.
- Reduces debugger use.
- (When you find - and fix - a new bug, write a test for it!)
- Don't forget to add some overall tests too.



I recommend giving TDD a try, but I'm still concerned as to where the overall design comes in. Rob Chatley will cover TDD in Software Engineering Design next year.