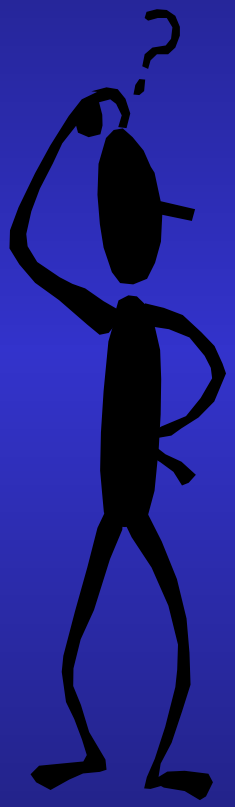
Unit Testing

what is a unit?

- A Unit Test is
  - Isolated – it has no external dependencies
    - it passes or fails *solely* due to the test code and the code under test
  - Repeatable
    - No external dependencies means if it passed last time and the code hasn't changed it will pass this time
  - Automatable
    - No external dependencies means it can run all by itself
  - Fast
    - No external dependencies means nothing external is slowing the test execution down

testability

- How to unit test one function…



example.c

```
#include <stdio.h>
...
void not_easily(void)
{
    fputc('4', stdout);
    fputc('2', stdout);
}
```

<stdio.h>

```
...

FILE * stdout = ...;

int fputc(int c, FILE * f)
{
    ...
}
```

step 1

- Move the function definition into its own file…
  - Include this file from where it used to live

example.c

```
#include <stdio.h>
...
#include "not_easily.func"
```

not_easily.func

```
void not_easily(void)
{
    fputc('4', stdout);
    fputc('2', stdout);
}
```

- Include the *source* in the test

not_easily.tests.c

```
#include <assert.h>

...
#include "not_easily.func"

void not_easily_tests(void)
{
    not_easily();
}
```

step 2

Create a mock environment

step 3

not_easily.tests.c

```
#include <assert.h>

typedef int mock_file;

static mock_file * stdout = 0;

static int fputc(int c, mock_file * f)
{
    ...
}

#include "not_easily.func"

void not_easily_tests(void)
{
    not_easily();
}
```

step 4

Test from within the mock environment

not_easily.tests.c

```
#include <assert.h>

typedef int mock_file;

static mock_file * stdout = 0;

static int fputc(int c, mock_file * f)
{
    static int count = 0;
    static int expected[2] = { '4', '2' };
    assert(expected[count] == c);
    count = (count + 1) % 2;
    ...
}

#include "not_easily.func"

void not_easily_tests(void)
{
    not_easily();
}
```

too brutal?

- Write a simple utility to read given lines (eg 311-355) from a named file and save these lines to a new file as part of the test build
  - Simple verification is probably good enough
    - First line contains the named function
    - Last line contains only a }
- Include this made file from the source instead of including the .func file
  - This way leaves the original source file completely intact

for example

get_lines.rb

```
lines = STDIN.readlines
start = ARGV[0].to_i - 1
finish = ARGV[1].to_i - 1
STDOUT.write lines[start..finish]
```

testability

- How to unit test dependent functions?

example.c

```
#include "local.h"
#include <string.h>
#include <stdio.h>
...
int b(int value)
{  ...
}

int c(int value)
{  ...
}

int a(int value)
{
      return b(value) + c(value);
}
```

Do includes indirectly

step 1

```
#define LOCAL(x)   #x
#define SYSTEM(x) <x>
```

example.c

```
#include LOCAL(local.h)
#include SYSTEM(string.h)
#include SYSTEM(stdio.h)

...

int a(int value)
{
    return b(value) + c(value);
}
```

step 2

- Divert includes when unit-testing

```
#ifdef UNIT_TESING

#   define LOCAL(x)   "mock/" ## #x
#   define SYSTEM(x)  LOCAL(x)

#else

#   define LOCAL(x)   #x
#   define SYSTEM(x)  <x>

#endif
```

step 3

- Include the *source* in the test

example.tests.c

```
#include <assert.h>

#define UNIT_TESTING
#include "example.c"

void example_test(void)
{
  assert(3 == a(42));
  ...
}
```

step 4

Create a mock environment

specific/mock/stdio.h

```
typedef int mock_file;

static mock_file * stdout = 0;

int fputc(int c, mock_file * f)
{
    ...
}
```

specific/mock/string.h

```
int strcmp(const char * lhs, const char * rhs)
{
    ...
}
```

specific/mock/local.h

```
...
```

step 5

- **Test from within the mock environment**

specific/mock/stdio.h

```
typedef int mock_file;

static mock_file * stdout = 0;

int fputc(int c, mock_file * f)
{
    assert(...);
}
```

specific/mock/string.h

```
int strcmp(const char * lhs, const char * rhs)
{
    assert(...);
}
```

too brutal?

- Write a simple utility copying input to new output file, except includes are commented
  - #include "abc.h" → /* #include "abc.h" */
  - #include <def.h> → /* #include <def.h> */
- Include this generated file in the test file
  - This leaves the original source file completely intact
  - This way all mock functions, mock data, and mock types can live inside the one test file

for example

excluder.rb

```
include  = Regexp.new('(\s*)#(\s*)include(.*)')

STDIN.readlines.each do |line|
  if m = include.match(line)
    line = "#if 0\n" + line + "#endif\n"
  end
  STDOUT.write line
end
```

```
cat wibble.c | ruby excluder.rb > wibble.isolated.c
```

wibble.tests.c

```
...
#include "wibble.ioslated.c"

void wibble_tests(void)
{
    ...
}
```

impossible?

- Code that is impossible to unit-test...
- Is not impossible to test
- It might be hard to unit test
- It might be awkward to unit test
- It might be messy to unit test
- But it *will* be possible

ask yourself...

- How hard are you willing to try?
- Only when you try will you find out...
- What makes code easy to unit test
- What makes code hard to unit test
- What tangled dependencies most code has

summary

- Writing unit tests takes effort
  - At first this effort is new and frightening
- That effort is rewarded many times over
  - It's never as bad as you think anyway
- Unit tests help you to write better code
  - You get better over time
- Unit tests create confidence to *refactor*
  - The codebase gets better over time
- Unit tests *improve* productivity
  - Why do cars have brakes?

books

- The C Programming Language
  - Kernighan and Ritchie
- The C Standard
  - BSI
- C: A Reference Manual
  - Harbison and Steele
- C FAQ's
  - Steve Summit
- Expert C Programming
  - Peter van der Linden
- Safer C
  - Les Hatton
- The Standard C Library
  - P.J.Plauger

chapter and verse!

legacy code

- ...

Michael Feathers
Working Effectively With Legacy Code

> ***Legacy code is code that has no tests.***
>
> ***A unit test that takes 1/10th of a second to run is a slow unit test.***
>
> ***Characterisation tests...***

- Advice on test output

Advanced C

Unit Testing

what is a unit?

- A Unit Test is
  - Isolated – it has no external dependencies
    - it passes or fails *solely* due to the test code and the code under test
  - Repeatable
    - No external dependencies means if it passed last time and the code hasn't changed it will pass this time
  - Automatable
    - No external dependencies means it can run all by itself
  - Fast
    - No external dependencies means nothing external is slowing the test execution down

testability

- How to unit test one function…

example.c

```
#include <stdio.h>
...
void not_easily(void)
{
    fputc('4', stdout);
    fputc('2', stdout);
}
```

<stdio.h>

```
...

FILE * stdout = ...;

int fputc(int c, FILE * f)
{
   ...
}
```

step 1

- Move the function definition into its own file…
  - Include this file from where it used to live

example.c

```
#include <stdio.h>
...
#include "not_easily.func"
```

not_easily.func

```
void not_easily(void)
{
    fputc('4', stdout);
    fputc('2', stdout);
}
```

step 2

- Include the *source* in the test

not_easily.tests.c

```
#include <assert.h>

...
#include "not_easily.func"

void not_easily_tests(void)
{
    not_easily();
}
```

step 3

- Create a mock environment

not_easily.tests.c

```
#include <assert.h>

typedef int mock_file;

static mock_file * stdout = 0;

static int fputc(int c, mock_file * f)
{
    ...
}

#include "not_easily.func"

void not_easily_tests(void)
{
    not_easily();
}
```

step 4

- Test from within the mock environment

not_easily.tests.c

```
#include <assert.h>

typedef int mock_file;

static mock_file * stdout = 0;

static int fputc(int c, mock_file * f)
{
    static int count = 0;
    static int expected[2] = { '4', '2' };
    assert(expected[count] == c);
    count = (count + 1) % 2;
    ...
}

#include "not_easily.func"

void not_easily_tests(void)
{
    not_easily();
}
```

too brutal?

- Write a simple utility to read given lines (eg 311-355) from a named file and save these lines to a new file as part of the test build
  - Simple verification is probably good enough
    - First line contains the named function
    - Last line contains only a }
- Include this made file from the source instead of including the .func file
  - This way leaves the original source file completely intact

for example

get_lines.rb

```
lines = STDIN.readlines
start = ARGV[0].to_i - 1
finish = ARGV[1].to_i - 1
STDOUT.write lines[start..finish]
```

testability

- How to unit test dependent functions?

example.c

```
#include "local.h"
#include <string.h>
#include <stdio.h>
...
int b(int value)
{ ...
}

int c(int value)
{ ...
}

int a(int value)
{
    return b(value) + c(value);
}
```

step 1

- Do includes indirectly

```
#define LOCAL(x)   #x
#define SYSTEM(x) <x>
```

example.c

```
#include LOCAL(local.h)
#include SYSTEM(string.h)
#include SYSTEM(stdio.h)

...

int a(int value)
{
    return b(value) + c(value);
}
```

step 2

- Divert includes when unit-testing

```
#ifdef UNIT_TESING

#  define LOCAL(x)   "mock/" ## #x
#  define SYSTEM(x) LOCAL(x)

#else

#  define LOCAL(x)   #x
#  define SYSTEM(x) <x>

#endif
```

step 3

- Include the *source* in the test

example.tests.c

```
#include <assert.h>

#define UNIT_TESTING
#include "example.c"

void example_test(void)
{
  assert(3 == a(42));
  ...
}
```

step 4

- Create a mock environment

specific/mock/stdio.h

```
typedef int mock_file;

static mock_file * stdout = 0;

int fputc(int c, mock_file * f)
{
   ...
}
```

specific/mock/string.h

```
int strcmp(const char * lhs, const char * rhs)
{
   ...
}
```

specific/mock/local.h

```
...
```

step 5

- **Test from within the mock environment**

specific/mock/stdio.h

```
typedef int mock_file;

static mock_file * stdout = 0;

int fputc(int c, mock_file * f)
{
    assert(...);
}
```

specific/mock/string.h

```
int strcmp(const char * lhs, const char * rhs)
{
    assert(...);
}
```

too brutal?

- Write a simple utility copying input to new output file, except includes are commented
  - #include "abc.h" → /* #include "abc.h" */
  - #include <def.h> → /* #include <def.h> */
- Include this generated file in the test file
  - This leaves the original source file completely intact
  - This way all mock functions, mock data, and mock types can live inside the one test file

for example

excluder.rb

```
include  = Regexp.new('(\s*)#(\s*)include(.*)')

STDIN.readlines.each do |line|
  if m = include.match(line)
    line = "#if 0\n" + line + "#endif\n"
  end
  STDOUT.write line
end
```

```
cat wibble.c | ruby excluder.rb > wibble.isolated.c
```

wibble.tests.c

```
...
#include "wibble.ioslated.c"

void wibble_tests(void)
{
    ...
}
```

impossible?

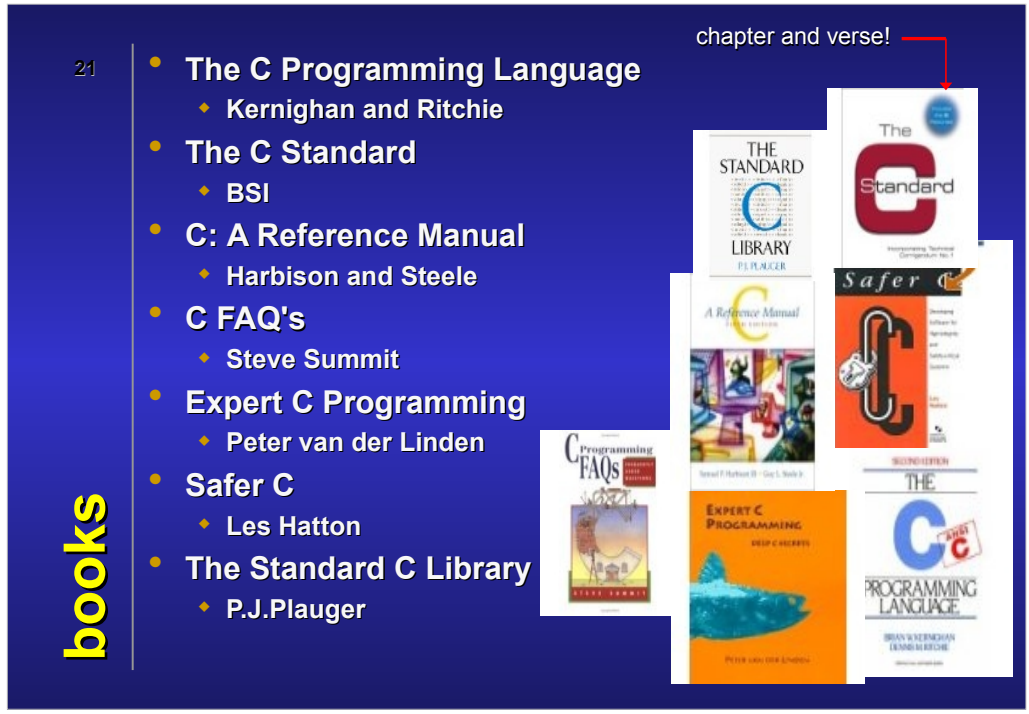
- Code that is impossible to unit-test...
- Is not impossible to test
- It might be hard to unit test
- It might be awkward to unit test
- It might be messy to unit test
- But it *will* be possible

ask yourself...

- How hard are you willing to try?
- Only when you try will you find out...
- What makes code easy to unit test
- What makes code hard to unit test
- What tangled dependencies most code has

summary

- Writing unit tests takes effort
  - At first this effort is new and frightening
- That effort is rewarded many times over
  - It's never as bad as you think anyway
- Unit tests help you to write better code
  - You get better over time
- Unit tests create confidence to *refactor*
  - The codebase gets better over time
- Unit tests *improve* productivity
  - Why do cars have brakes?



Also worth knowing about is a free PDF book on the C Standard written by Derek Jones, one of the world's foremost experts on C:

http://www.knosof.co.uk/cbook/cbook.html

This book contains a commentary on every single sentence in the C Standard. It is invaluable if you are reading the C Standard and need some help understand what specific sentence means.

legacy code

- …

Michael Feathers
Working Effectively With Legacy Code

> ***Legacy code is code that has no tests.***
>
> ***A unit test that takes 1/10th of a second to run is a slow unit test.***
>
> ***Characterisation tests...***

- Advice on test output