Doing C/C++ Unit Testing on a Shoestring

By Ark Khasin, MacroExpressions
akhasin@macroexpressions.com

Purpose of the class

- To help you make rational decision in selecting your testing system
- To propose a novel approach to code instrumentation

Outline of the class

- A broad look at commercial and free test automation tools
- To propose a framework for unit testing
- To see how much test automation can be done with it for free
  - To discuss limitations of the proposed method
  - To see what a coding style / coding standard can support

The purpose of unit testing

- Demonstrate correctness of the code to yourself and to the auditor
  - your tests passed, and
  - your tests sufficiently cover the function's implementation
- Ensure your code is still correct after modifications (Regression testing)
- Even a "proven in use" function may fail on a previously masked execution path
  - my bug in a string output function
  - "Which brings us to tonight's word:" Coverage

Common Types of Coverage

"Condition" means an atomic Boolean expression in a compound Boolean "decision" expression in `if`, `while`, `do`/`while` and `for`.

- Branch coverage (a.k.a. decision coverage)
- Condition coverage: each condition has been true and false at least once
- Condition/Decision (CD) coverage – a union of condition and branch coverage
- Much touted Modified Condition/Decision (MC/DC) coverage – CD coverage where each condition is shown to affect decision independently
  - If the (atomic) conditions are not independent, you won't get MC/DC coverage.
  - But you can get away with Recursive MC/DC (below)
- Boundary conditions coverage

Process of Unit Testing

- Define a test set for the UUT
  - Add, Remove or Modify test cases
  - Define acceptance criteria for each test
- Execute the tests
  - With the sole purpose to produce test results
- Evaluate test results
  - And fix the code as needed and repeat
- Evaluate achieved coverage
  - Add test cases as needed and repeat

To do the execution step,

- Some functions may need to be “stubbed out” to replace the real implementations, which
  - May not be yet available
  - May not be essential to the test

To do the evaluation steps,

- The UUT must be “instrumented” to produce more output than normally available
- Instrumentation puts a limit on
  - what tests you can design, and
  - what test coverage you can achieve

Relation between test execution and evaluating results

A non-instrumented test output

- CT-scan test output is analyzed by a qualified doctor
- Software test output is analyzed by a qualified software engineer aided by appropriate analysis software

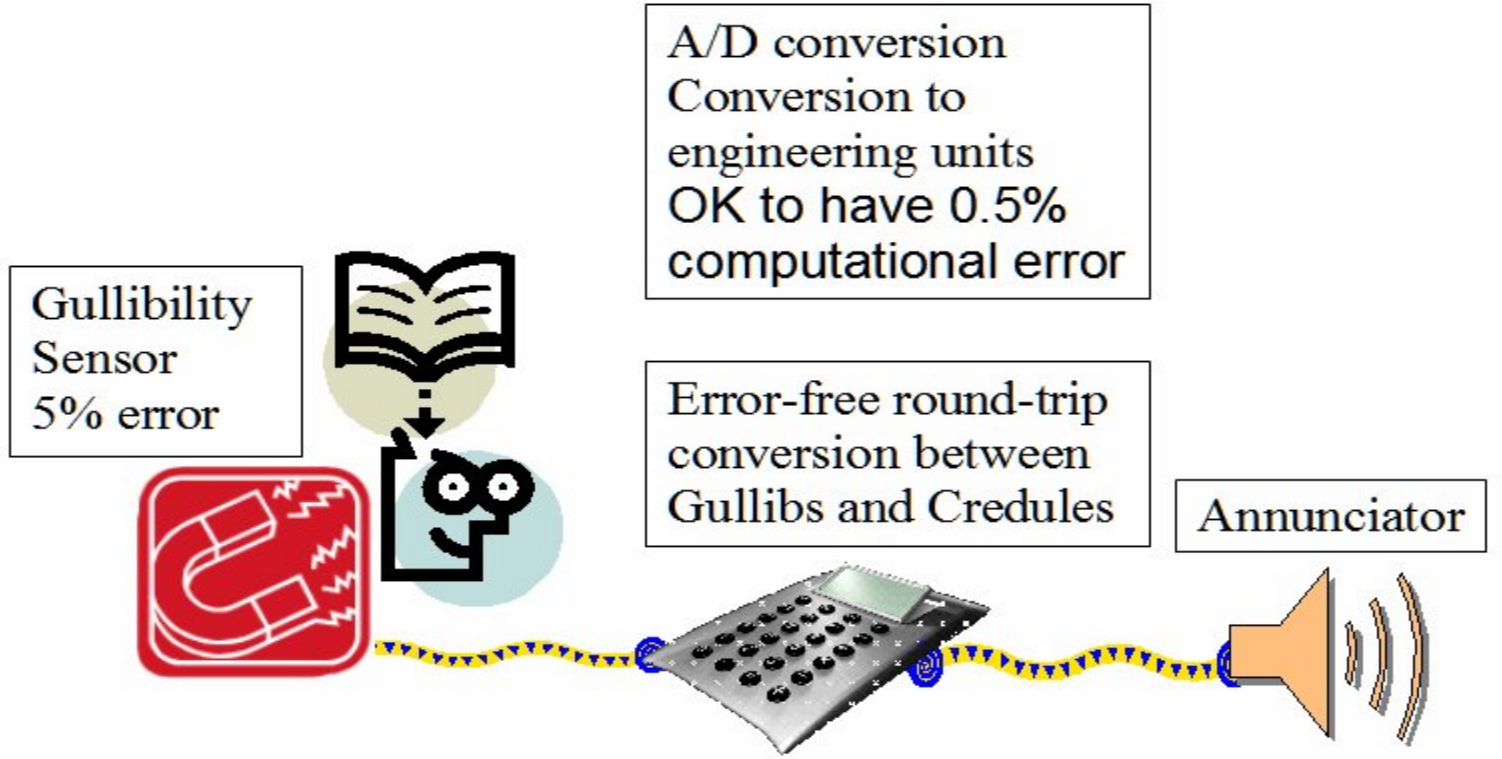
Add Instrumentation

- Produces more output than normally visible
- Contrast CT-scan requires “instrumenting” your body’s organs to produce more output without changing their essential behaviors
  - Or, at least, such is the belief
- Software testing requires instrumenting your code to produce more output without changing its essential behavior
  - Or, at least, such is the belief
- Verbose execution trace
  - To support regression testing
  - To support Test Output Analysis

Let's take a closer look at the process with an eye on possible automation.

Test cases

- Automation tools may suggest certain test cases
  - Primarily, targeted at specific models of test coverage
- Don't expect automation tools to understand your design
  - A fair amount of test cases must be created manually
  - Example:



Stubs

- You have to decide what to stub
  - stubbing out `strcpy()` might not be a smart idea
  - stubbing out a function to which creating some side effects is usually desirable
- Automation tools aren't smart enough
  - may offer to stub all or none, or
  - leave you with DIY which is usually the best

Harness (execution environment)

- In a good DIY design, you implement it once
- Expect it fully automated by a vendor

Instrumentation

- Can be hardware- and/or software-based
- For unit testing, purely software instrumentation is normally used
  - Can and must be automatic for repeatability and error elimination
  - Bugs in instrumentation are typically catastrophic (may lead to incorrect test result evaluation)

Free unit testing frameworks

- Free
- Open-source
- Provide harness
- Manual creation of test cases
- No instrumentation
- No proof of coverage
- Lump test execution with test management (large footprint)
- Require compiler/CPU adaptation
- Depend on dynamic memory management

Commercial unit testing frameworks

- Expensive
- Proprietary code
- Provide harness
- Suggest test cases
- Provide instrumentation based on parsing the code
- Provide coverage analysis
- Split test execution, analysis and management
- Require compiler/CPU adaptation
- Generally produce small footprint
- Bugs seen in instrumentation or coverage, which is catastrophic

Proposed unit testing framework

- DIY with free reference implementation
- Open Source reference implementation
- Manual creation of test cases
- Focused on instrumentation
- Automatically adapts to your compiler/CPU
  - Based on abusing C/C++ preprocessor
- Has limitations
  - Some may be addressed by a company coding standard
    - E.g. based on MISRA C
- Opens an opportunity for compiler-independent instrumentation
  - Is anyone interested?

On to the Unit Test Framework

A taste of code instrumentation

- Goal: gain access to `static` variables in UUT
- Begin abusing the preprocessor:
  `#define static extern`
- Turns *uninitialized* definitions of internal linkage into declarations of external linkage
- Turns *initialized* definitions of internal linkage into definitions of external linkage
  - Therefore, breaks initialized `static` in block scope
    - which may be a good thing
- Doesn't work with C/C++ block name resolution
  - but it should be proscribed by coding standard in a safety-related environment
- Harness requirements:
  - Need definitions for new declarations
    `static T x;` in the UUT requires
    `T x;` in the harness
  - Need declarations for new definitions
    `static T x = X;` in the UUT requires
    `extern T x;` in the harness

Where to put this abuse?

- A place readily available when you need it
- A place that guarantees non-interference with the normal build
- Solution: invent a header "`instrum.h`" as a general-purpose wrapper and "`instrum_uut.h`" for instrumenting your `uut.c` as follows:

`instrum.h`

```
#ifdef INSTRUM_HEADER

#define INSTRUM_STATIC

/*request abuse of other
keywords*/
............

#include INSTRUM_HEADER
#endif /*INSTRUM_HEADER*/
```

`instrum_uut.h`

```
//prototypes
#include "instrum_implem.h"
//abuse of static
#ifdef INSTRUM_STATIC
# define static extern
#endif /*INSTRUM_STATIC*/

//abuse of other keywords
............
```

Technique 1: If you have a common header, say, `project.h`

project.h (original)

```
//common goodies
#include <stddef.h>
............

//common project definitions
#define FOO 1234567
............
```

project.h (updated)

```
//common goodies
#include <stddef.h>
............

//Magic incantation
#include "instrum.h"

//common project
definitions
#define FOO 1234567
............
```

Technique 2: Include the UUT in the harness file

- Additionally, create an "un-instrumentation" header `uninstrum.h` to undefine some of the abuses created for instrumentation, e.g.:
  ```
  #undef static
  ```
- Begin your harness with
  ```
  #include "instrum.h"
  #include "uut.c"
  #include "uninstrum.h" //remove instrumentation
  ```
- Note: instrumenting the `static` keyword is unnecessary for this technique
  - Except if you want to break the block-scope `static`.

For either technique,

- To get instrumented compilation,
  - Add definition (usually, -D flag) to the compiler command line:
    ```
    INSTRUM_HEADER="instrum_uut.h"
    ```
  - Don't forget to escape quotes according to your shell

A note on stubs

- Occasionally, you want a stub for `foo()` to call the real implementation of `foo()`
- Can be done with a special version of `instrum_uut.h`, say, `instrum_myown.h`
  - So, pass to the compiler
    `INSTRUM_HEADER="instrum_myown.h"`
  - Requires that `foo()` be defined outside the UUT
- Name the stub differently, e.g. `stub_foo()`

instrum_myown.h

```
//common instrumentation
#include "instrum_uut.h"

//fool the UUT by renaming
#define foo stub_foo

//(optional) prototype for foo()
// - turns into prototype for stub_foo
#include "foo.h"
```

- Implement `stub_foo()` elsewhere as you normally would

C Code Instrumentation

`if` keyword

- Put in `instrum.h`

```
#define INSTRUM_IF
```

- Add to `instrum_uut.h`

```
#ifdef INSTRUM_IF
#define if(condition) \
  if(instrum_if(#condition, (condition)!=0, \
  __FILE__, __LINE__, __FUNCTION__))
#endif
```

- Add to `instrum_implem.h`

```
extern int instrum_if(const char *condition_name,
                      int condition,
                      const char *filename,
                      int line,
                      const char *function_name);
```

`if` keyword (cont'd)

- Add implementation of `instrum_if`, e.g.

```
int instrum_if(const char *condition_name,
               int condition,
               const char *filename,
               int line,
               const char *function_name)
{
    printf("Condition %s in funtion %s"
        " (file %s line %d) is %s\n",
        condition_name, function_name,
        filename, line, (condition)?"true":"false");
    return condition;
}
```

- Note: an implementation of `instrum_if` must return the value of the parameter `condition`
- Note: a useful implementation of `instrum_if` is more involved because instrumentation of other keywords depend on `if`

`if` keyword (cont'd)

Code coverage support

- If control flow is implemented with `if/else` only, 100% branch coverage is achieved if every `if` had its condition at least once true and at least once false
- Can be discovered by parsing the instrumented test output provided every if can be uniquely identified

Limitations

- Constructs like
  `if(++x) {a}; if(++x) {b}`
  make it hard to implement uniquely identifiable `if` instrumentation
- In most cases, can be addressed by a Coding Policy
- Can be a problem if a result of a macro expansion
  - But should those macros be considered opaque and tested separately?!

`while` keyword

- Can follow the pattern of simplistic instrumentation of `if`
- Optional special treatment of idioms `while(0)` and `while(1)`, e.g.

```
int instrum_while(const char *condition_name,
                  int condition,
                  const char *filename,
                  int line,
                  const char *function_name)
{
   if(strcmp(condition_name, "0") != 0 &&
      strcmp(condition_name, "1") != 0) {
            printf("Condition %s in funtion %s"
                   " (file %s line %d) is %s\n",
                condition_name, function_name,
                filename, line, (condition)?"true":"false");
   }
   return condition;
}
```

- Same Coverage and Limitations considerations apply

`switch` keyword

- Can follow the pattern of simplistic instrumentation of `if`
- Useful for regression testing but by itself little help with coverage
- Coverage proof requires instrumenting `case` and `default` (next)

`default` keyword

- Can follow the pattern of instrumentation of `static`
- Based on the observation that
  `default:`
  is equivalent to
  `default: if(some_true_value) a_unique_label:`
- For `some_true_value`, we take (the return value of) the function `instrum_default` which must return a non-zero
- For `a_unique_label` we can take a concatenation of `instrum_label` and *expanded* macro `__LINE__`

`default` keyword (cont'd)

- Here is a working definition

```
#define default \
    default: \
    if(instrum_default(__FILE__,__LINE__,__FUNCTION__)) \
        CAT(instrum_label, __LINE__)
```

  where `CAT` concatenates two *expanded* tokens:

```
#define CAT1(a,b) a ## b
#define CAT(a,b) CAT1(a,b)
```

- Note: the `if` comes in already instrumented; we must suppress announcements from it. That's why the `if` instrumentation is more involved.

Coverage support

- A `default` statement was covered if it was announced (by `instrum_default`)

Limitations

- One `default` in a line of code

`case` keyword

- Instrumentation requires parenthesizing the label: Instead of
  `case MYCASE:`
  write
  `case (MYCASE):`
- Can be required by a Coding Standard but is not common and can be considered intrusive
- Implementation can follow the pattern of `default` instrumentation

Coverage support

- If a `case` was announced (by `instrum_case`) it was covered

Limitations

- One instrumented `case` in a line of code
- A numeric label must be parenthesized (or a `case` won't be instrumented)

Difficult cases

- Control expression of a `for` loop
- The ternary operator `?:`
- Bizarre coding practices, e.g. `!a || b` instead of `if(a) b`;
  - May have merits in macro definitions
- Solution possible but is intrusive:
  - `ISTRUE` macro
    - in `instrum.h`, a definition for normal build:
      ```
      #ifndef ISTRUE
      #define ISTRUE(e) (e)
      #endif
      ```
    - in `instrum_uut.h`
      ```
      #define ISTRUE(e) \
      instrum_istrue(#e,(e),__FILE__,__LINE__,__FUNCTION__)
      ```
      with appropriate definition of `instrum_istrue`.
    - Require that non-empty control expressions of `for` loops and non-constant ternary operators be wrapped in `ISTRUE`

Difficult cases (cont'd)

- Condition coverage support (cannot redefine `||` or `&&`)
  - Use `and` for `&&`, `or` for `||` etc.
    - Built into C++; C requires
      `#include <iso646.h>`
    - Parenthesize logical expressions (not a bad practice)
    - `#undef` and redefine the logical operators in `instrum_uut.h`, e.g.

```
#define and(e) \
&& instrum_and(#e, (e)!=0, __LINE__)
```

      where `instrum_and` returns the value of the second parameter (and similarly for other operations)

Other keywords

- `break`, `continue`, `goto`, `return`
- Any **`keyword`** among them is functionally equivalent to a passage
  `if(false_value) {exit(1);} else keyword`
- Following the common pattern,

```
#define keyword \
if((instrum_keyword(__FILE__,__LINE__,__FUNCTION__), \
instrum_false)) {for(;;);} else keyword
```

Abnormal control flow

- No special instrumentation needed for `setjmp`/`longjmp`
- (C++ only) Probably `try/throw/catch` cannot be instrumented in a useful way
  - Except that `throw` can be instrumented like `return`.

Coverage Analysis

Branch Coverage

- If all program flow is controlled by `if/else`,
  - 100% coverage is achieved if every (uniquely identified) `if` statement announced its condition to be true and to false at least once each
- If the ternary operator is added to control flow,
  - 100% coverage is achieved if, in addition, it announces (via `ISTRUE`) its control expression to be true and to false at least once each
    - You get to decide which ternary operator is considered affecting program flow
- If `do/while` and/or `while` are added to program flow control,
  - 100% coverage is achieved if, in addition, all `while` conditions are announced true and false at least once each
    - Provided you took care of `while`-related idioms
- If `for` is added to program flow control,
  - 100% coverage is achieved if, in addition, it announces (via `ISTRUE`) its control expression to be true and to false at least once each
    - And nothing interesting about `for(expr1; ; expr3)`
    - You are responsible for wrapping the control expression in `ISTRUE`

Branch Coverage (cont'd)

- If `switch/case/default` is added to program flow control,
  - 100% coverage is achieved if, in addition,
    - for each announced `switch`, each `case` and `default` is at least once announced as the first case, or for a `switch` without `default`, at least once there was no announcement
    - You are responsible for enclosing case labels in parentheses
- If `goto` is added to program flow control,
  - The scheme falls apart. Do no use `goto`.
    - Instrument `goto` to fail the test

Condition Coverage

- MC/DC for logical terms connected with only `and` or only `or` is easily supported.

Example:

```
if( (a) or (b) )
```

where `a` and `b` are logical expressions.
In execution of instrumented code (of `if` and `or` are instrumented!),

- `(a) or (b)` is evaluated
- `if` is announced as true or false (Decision)

In the first step:
If `(a)` was true, `or(b)` is not evaluated and not announced
If `(a)` was false, `or(b)` is evaluated and announced as true or false
You get CD coverage w.r.t. `a` and `b` if you have test cases where

- `or(b)` was not announced
- `or(b)` was announced false
- `or(b)` was announced true

Recursive MC/DC

- If `a` and/or `b` are in turn compound expressions, use this scheme recursively.
- Circumvents code constraint of MC/DC

Example: `a` is `(c) and (d)`.
Assume `a` satisfied MC/DC
When `(a)` is evaluated, `(c)` is evaluated first

- if `c` is false, `and(d)` is not evaluated and evaluation of `d` (and `a`) is not announced
- if `c` is true, `and(d)` is evaluated and is announced as true or false

Putting it all together

Test Execution Framework

Test Harness

Standard DYI code iterating over the table of
Description
Acceptance criteria
Test setup code (optional)
A number of *test cases*
Test cleanup code (optional)

Test case

Description (optional)
Parameters (optional)
The number of repetitions
Test case execution code (which actually exercises the function you're testing)

Stubs

Entirely DYI

Instrumentation

Based on `instrum_uut.h`
May be replaced with your UUT-specific `instrum_myown.h` via `INSTRUM_HEADER` macro

Reference implementation

http://www.macroexpressions.com/maestra.html