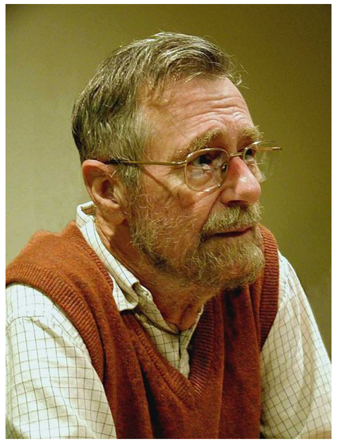
Laboratorio di Tecnologie dell'Informazione

Ing. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

Code testing: techniques and tools

"Testing can show the presence of errors, but not their absence."
- Edsger Dijkstra



What is software verification ?

- Software verification is a discipline of software engineering whose goal is to assure that software fully satisfies all the expected requirements.
- There are two fundamental approaches to verification:
  - Dynamic verification, also known as Test or Experimentation
    - This is good for finding bugs
  - Static verification, also known as Analysis
    - This is useful for proving correctness of a program although it may result in false positives

Static verification

- Static verification is the process of checking that software meets requirements by doing a physical inspection of it. For example:
  - Code conventions verification
  - Bad practices (anti-pattern) detection
  - Software metrics calculation
  - Formal verification

Dynamic verification

- Dynamic verification is performed during the execution of software, and dynamically checks its behaviour; it is commonly known as the Test phase. Verification is a Review Process. Depending on the scope of tests, we can categorize them in three families:
  - Test in the small: a test that checks a single function or class (Unit test)
  - Test in the large: a test that checks a group of classes
  - Acceptance test: a formal test defined to check acceptance criteria for a software

Static and Dynamic

- Static and Dynamic analysis are complementary in nature
    - Static unit testing is not an alternative to dynamic unit testing, or vice versa.
- It is recommended that static unit testing be performed prior to the dynamic unit testing

Static program analysis

- Static program analysis is the analysis of computer software that is performed without actually executing programs
- The term is usually applied to the analysis performed by an automated tool
    - code review is a human analysis procedure in which different programmers read the code and give recommendations on how to improve it.

Static program analysis - cont.

- It is possible to prove that (for any Turing complete language, like C and C++), finding all possible run-time errors in an arbitrary program (or more generally any kind of violation of a specification on the final result of a program) is undecidable...
- ...but one can still attempt to give useful approximate solutions

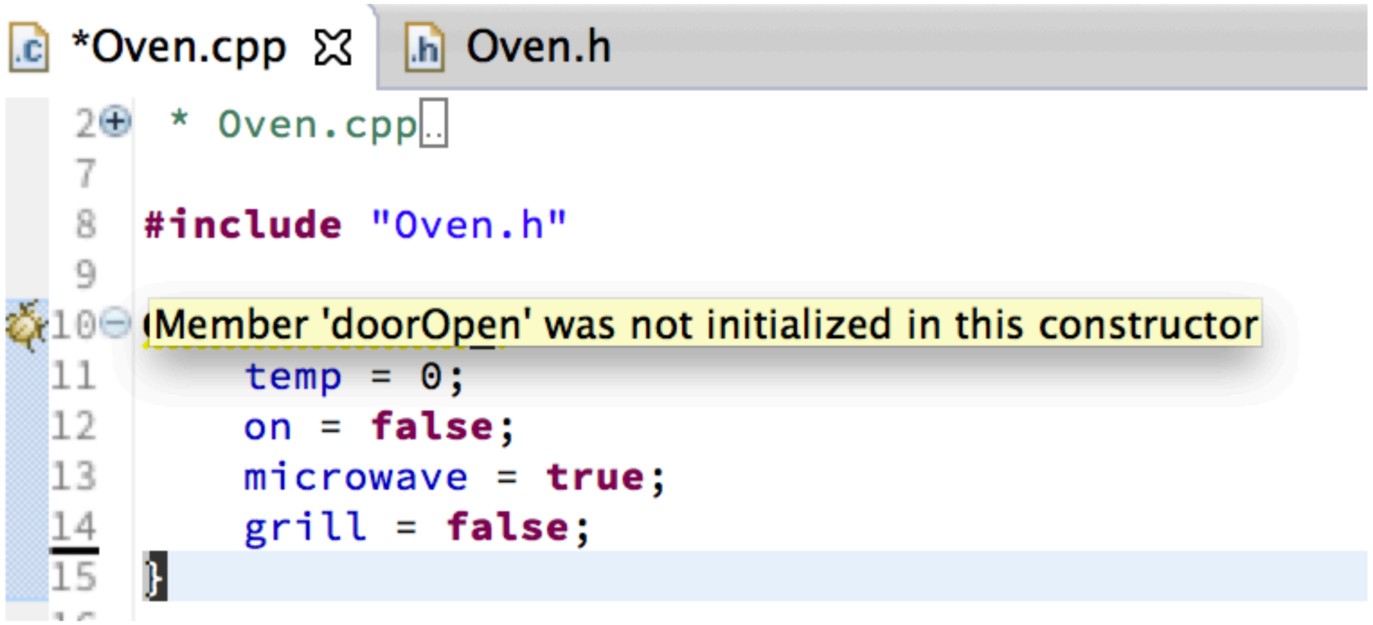
Static program analysis - cont.

- Many IDEs (e.g. Eclipse, XCode) have a static analysis component: CODAN in Eclipse, CLANG in XCode
- Other tools are available as plugins or standalone tools (e.g. cppcheck)

Eclipse shows results of its static analysis tools with bug icons... these are not results of the compiler !

Static program analysis - cont.

- Many IDEs (e.g. Eclipse, XCode) have a static analysis component: CODAN in Eclipse, CLANG in XCode

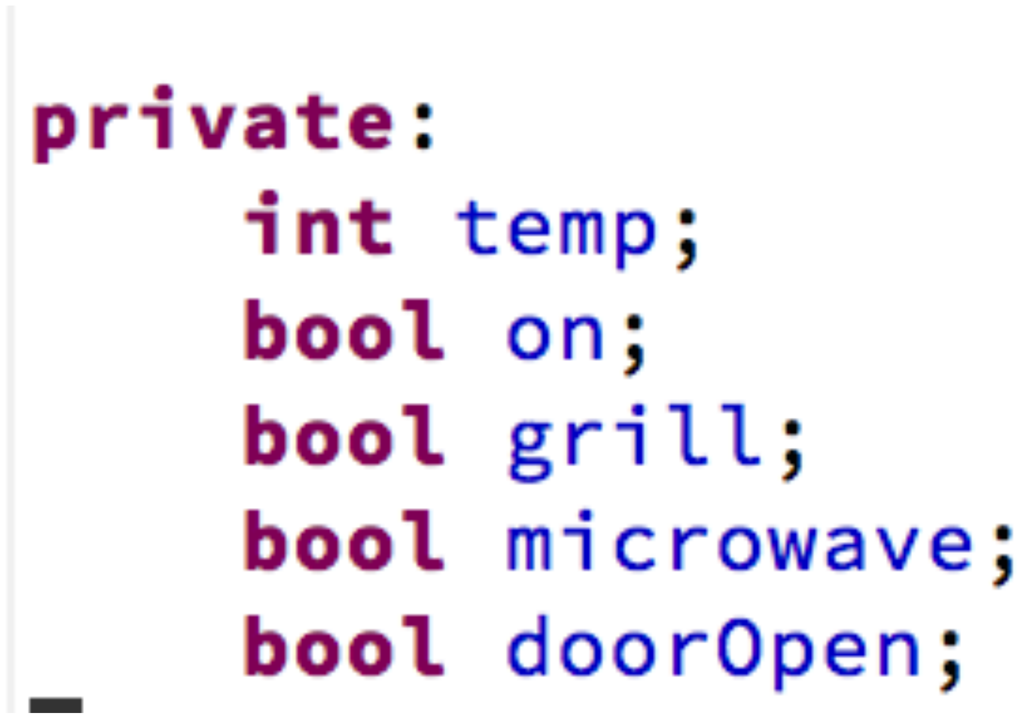


```
 * Oven.cpp

#include "Oven.h"

Member 'doorOpen' was not initialized in this constructor
    temp = 0;
    on = false;
    microwave = true;
    grill = false;
}
```

- Other tools are available as plugins or standalone tools (e.g. cppcheck)

Static program analysis - cont.

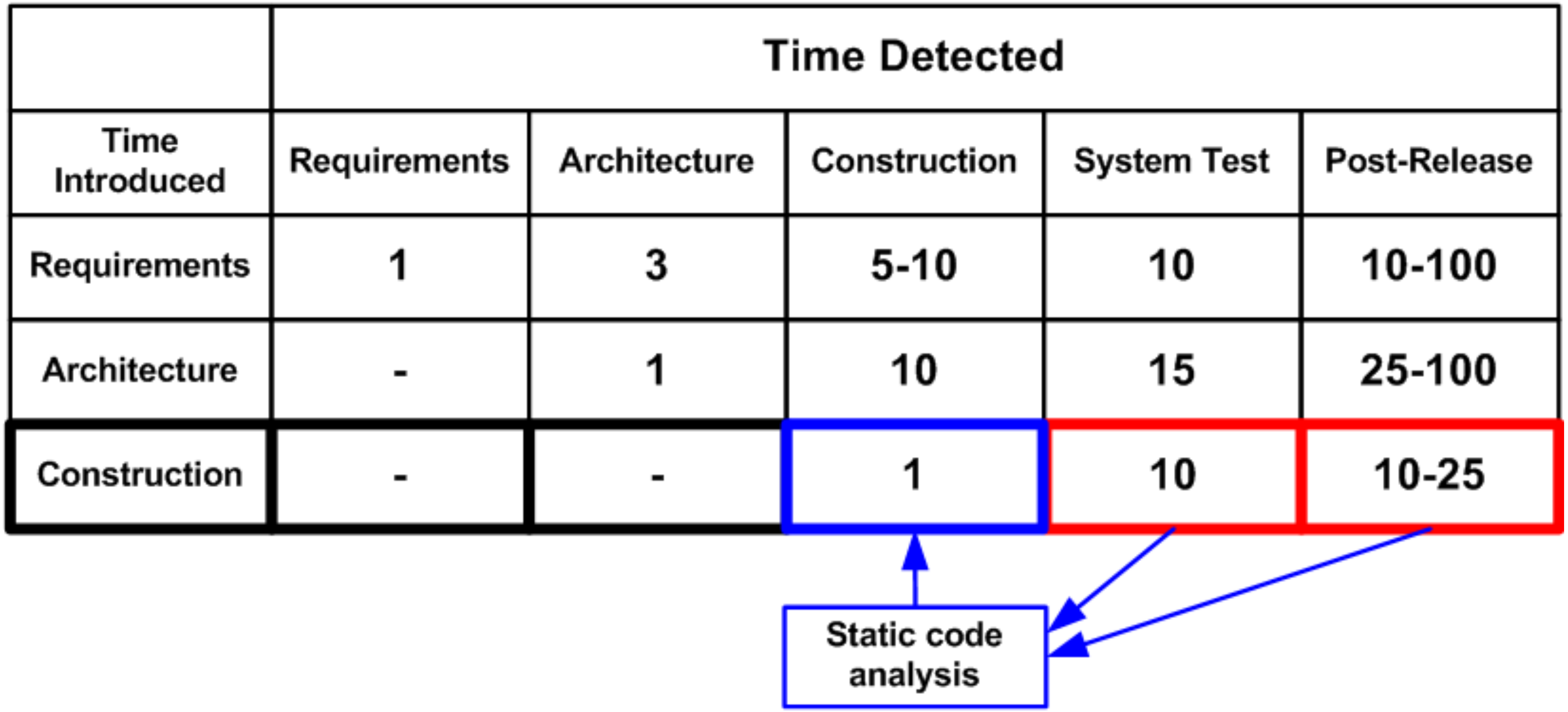
- Many IDEs (e.g. Eclipse, XCode) have a static analysis component: CODAN in Eclipse, CLANG in XCode
- Other tools are available as plugins or standalone tools (e.g. cppcheck)



```
private:
        int temp;
        bool on;
        bool grill;
        bool microwave;
        bool doorOpen;
```

Why using static analysis tools ?

- McConnel has reported in “Code Complete” that the cost of fixing an error at testing stage is 10x that of code writing stage:



	Time Detected				
Time Introduced	**Requirements**	**Architecture**	**Construction**	**System Test**	**Post-Release**
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	-	-	1	10	10-25

Static code analysis

Why using static analysis tools ?

- Static analysis tools allow you to quickly detect a lot of errors of the coding stage
- Static analyzers check even those code fragments that get controlled very rarely.
- Static analysis doesn't depend on the compiler you are using and the environment where the compiled program will be executed. It allows you to find hidden errors that can reveal themselves only a few years later.
- You can easily and quickly detect misprints and consequences of Copy-Paste usage.

Static code analysis' disadvantages

- Static analysis is usually poor regarding diagnosing memory leaks and concurrency errors. To detect such errors you actually need to execute a part of the program virtually.
  - There are specific tools for this, like Valgrind, AddressSanitizer and MemorySanitizer
- A static analysis tool warns you about odd fragments that can actually be quite correct. Only the programmer can understand if the analyzer points to a real error or it is just a false positive.

Unit testing

"A QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999 beers. Orders a lizard. Orders -1 beers. Orders a...."
- Amos Shapira, Senior System Engineer at Wargaming.net

Unit testing

- Unit testing is a method by which individual units of source code are tested to determine if they are fit for use.
  - It is NOT an academic distraction of exercising all inputs... it's a common practice in agile methods
  - It is becoming a substantial part of software development practice, with many frameworks and tools to help its implementation

Unit testing - cont.

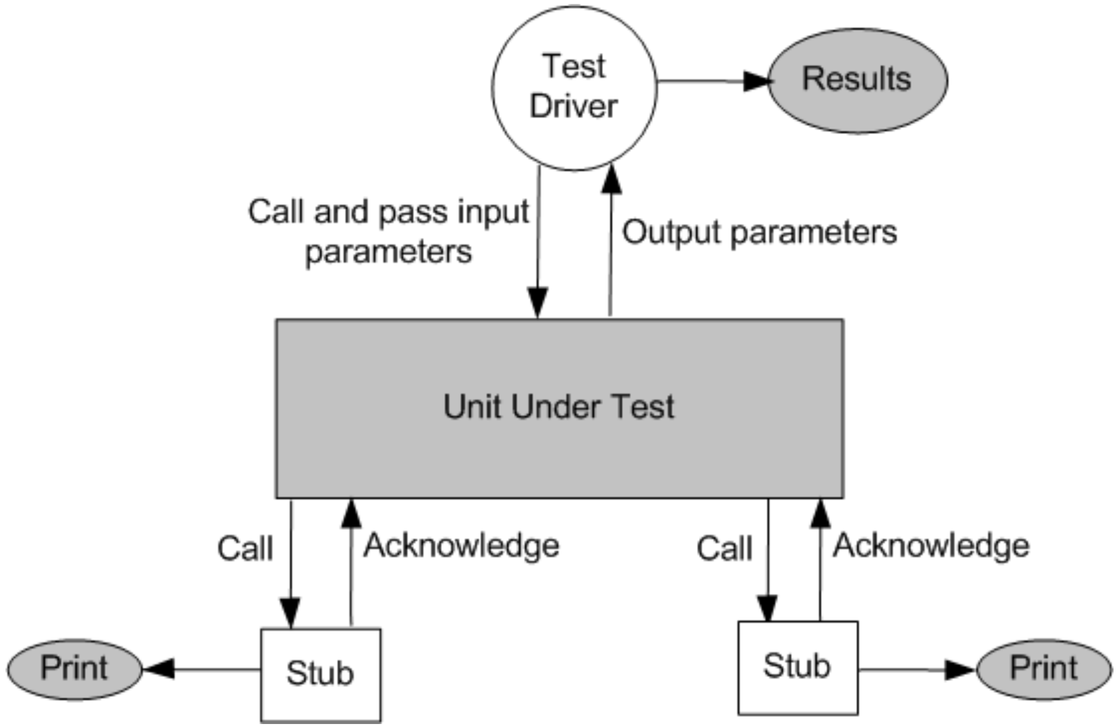
- The idea about unit tests is to write test cases (i.e. code) for all functions and methods so that whenever a change causes a problem, it can be identified and fixed quickly.
  - Ideally each test is separate from the others.
- To ease the task of writing the testing code you can use several frameworks like CppUnit, CxxTest, GoogleTest, Boost::Test, etc.

Unit testing - cont.

- Programmers create test inputs using intuition and experience, following the specifications
- Programmers determine proper output for each input using informal reasoning or experimentation
- A test run shows results like passed vs. failed tests, either onscreen or as a stream (in XML, for example). This latter format can be leveraged by an automated process to reject a code change.
- Ideally, unit tests should be incorporated into a makefile or a build process so they are ran when the program is compiled.

Unit testing - cont.

- The environment of a unit is emulated and tested in isolation
- The caller unit is known as test driver
    - A test driver is a program that invokes the unit under test
    - It provides input data to unit under test and report the test result
- The emulation of the units called by the UUT are called stubs
    - It is a dummy program
- The test driver and the stubs are together called **scaffolding**



Unit test: terms

- A test suite groups test cases around a similar concept. Each test case attempts to confirm domain rule assertions.
- There are situations where a group of test cases need a similar environment setup before running. A **test fixture**, enables a **setup** definition to be executed before each of its test cases. The fixture can include a **teardown** function to clean up the environment once a test case finishes, regardless of its results.

Unit test: example

```
 class Calculator {

public:
   Calculator();
   Calculator(int v);

   int sum(int x);
   int subtract(int x);
   int mul(int x);

   int getValue() const {
      return value;
   }

   void setValue(int value){
      this->value = value;
   }

private:
   int value;
};
```

Unit test: CxxTest example

```
#include <cxxtest/TestSuite.h>

#include "Calculator.h"

class TestCalculator : public
CxxTest::TestSuite {
public:
    void setUp() {
        // TODO: Implement setUp() function.
    }

    void test_global_Calculator_Calculator() {
        Calculator c;
        TS_ASSERT_EQUALS(c.getValue(), 0);
    }

    void
test_global_Calculator_Calculator_int() {
        Calculator c(3);
        TS_ASSERT_EQUALS(c.getValue(), 3);
    }

    void testSum() {
        Calculator c(4);
        c.sum(3);
        TS_ASSERT_EQUALS(c.getValue(), 7);
    }

    void testSubtract()
    {
        Calculator c(5);
        c.subtract(3);
        TS_ASSERT_EQUALS(c.getValue(), 2);
    }

    void testMul()
    {
        Calculator c(7);
        c.mul(2);
        TS_ASSERT_EQUALS(c.getValue(), 14);
    }

};
```

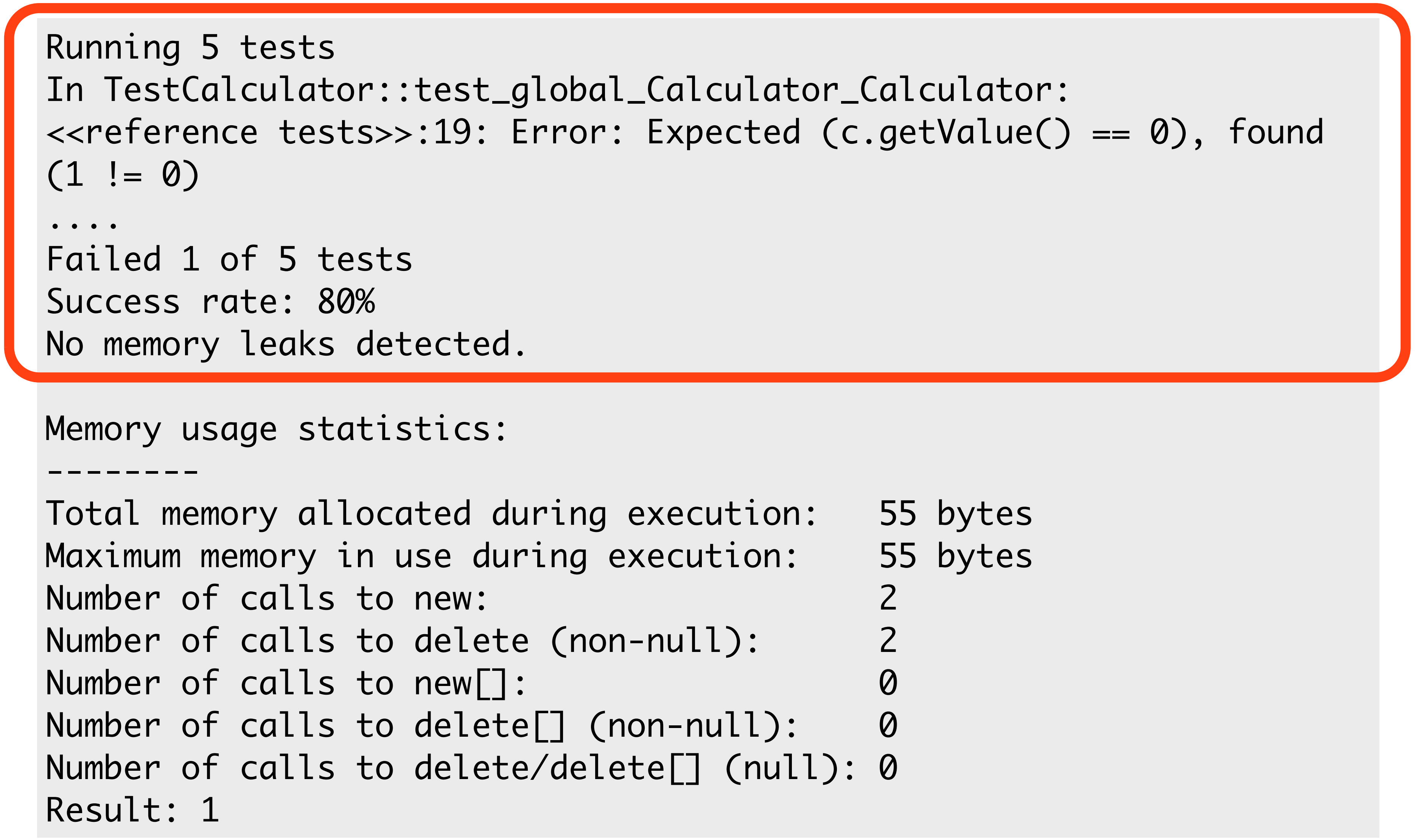
Unit test: CxxTest example

- Let’s suppose that the implementation of the constructor is wrong, here’s the output of the tests:

```
Running 5 tests
In TestCalculator::test_global_Calculator_Calculator:
<<reference tests>>:19: Error: Expected (c.getValue() == 0), found
(1 != 0)
....
Failed 1 of 5 tests
Success rate: 80%
No memory leaks detected.

Memory usage statistics:
---------
Total memory allocated during execution:   55 bytes
Maximum memory in use during execution:    55 bytes
Number of calls to new:                    2
Number of calls to delete (non-null):      2
Number of calls to new[]:                  0
Number of calls to delete[] (non-null):    0
Number of calls to delete/delete[] (null): 0
Result: 1
```

Unit test: CxxTest example

- Let’s suppose that the implementation of the constructor is wrong, here’s the output of the tests:



```
Running 5 tests
In TestCalculator::test_global_Calculator_Calculator:
<<reference tests>>:19: Error: Expected (c.getValue() == 0), found
(1 != 0)
....
Failed 1 of 5 tests
Success rate: 80%
No memory leaks detected.

Memory usage statistics:
---------
Total memory allocated during execution:   55 bytes
Maximum memory in use during execution:    55 bytes
Number of calls to new:                    2
Number of calls to delete (non-null):      2
Number of calls to new[]:                  0
Number of calls to delete[] (non-null):    0
Number of calls to delete/delete[] (null): 0
Result: 1
```

TDD

- In test-driven development (TDD), programmers first write the test code, then the actual source code, which should pass the test.
- This is the opposite of conventional development, in which writing the source code comes first, followed by writing unit tests, but...
- ...a recent study at Microsoft has shown that:
  - TDD teams produced code that was 60 to 90 percent better in terms of defect density than non-TDD teams.
  - TDD teams took longer to complete their projects—15 to 35 percent longer.

Debugging

- The process of determining the cause of a failure is known as debugging
- It is a time consuming and error-prone process
- Debugging involves a combination of systematic evaluation, intuition and a little bit of luck
- The purpose is to isolate and determine its specific cause, given a symptom of a problem
- IDEs like XCode and Eclipse integrate an external debugger (lldb or gdb). Install it.

Debugging

- The process of determining the cause of a failure is known as debugging
- It is a time consuming and error-prone process
- Debugging involves a combination of systematic evaluation, intuition and a little bit of luck
- The purpose is to isolate and cause, given a symptom of a pr
- IDEs like XCode and Eclipse integrate an external debugger (lldb or gdb). Install it.

To debug using Eclipse on OSX install gdb using Macports or Homebrew