CS 50:
Software Design and Implementation

Software testing

Questions

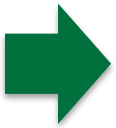
What is the difference between a bug and a vulnerability?

Can software bugs or vulnerabilities cause real-world harm?

Question: Can software bugs or vulnerabilities cause real world harm?

https://www.youtube.com/watch?v=fJyWngDco3g

Agenda



1. Types of tests
2. Unit test
3. Activity

Questions

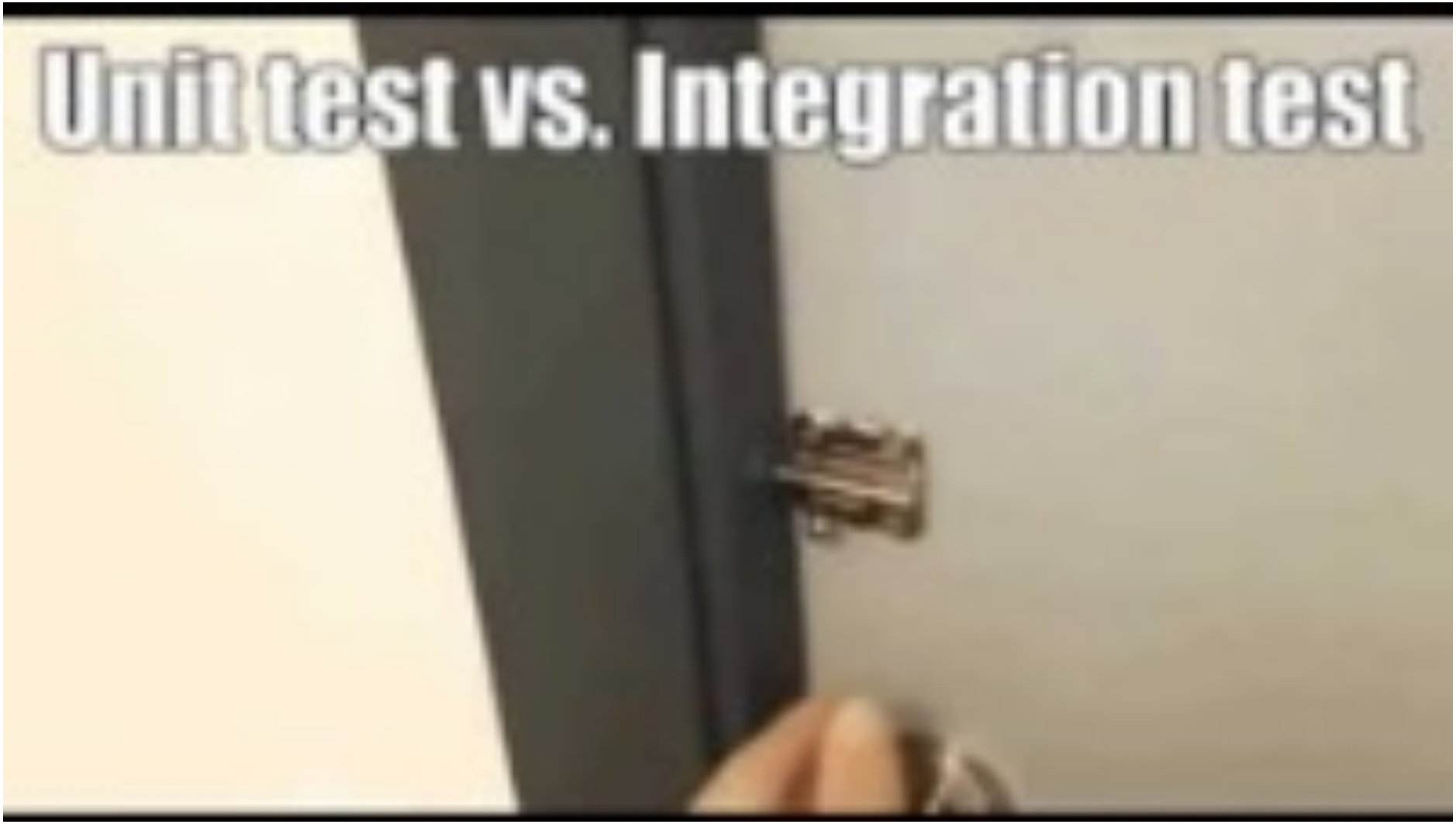
What is the difference between testing and debugging?

What kind of testing are you familiar with?

- Glass box vs black box
- Unit
- Functional
- Integration
- System
- Regression
- Usability
- Security
- Fuzz
- Acceptance

Beware of "scope creep"
Or it's neighbor "can't you just do this on the side?"
(especially during functional testing)

Unit tests don’t always reveal problems! Integration testing is also important



https://www.youtube.com/watch?v=0GypdsJulKE

Testing tips

1. Test incrementally and build confidence in your code
2. Write unit tests that can be re-run once fixes or changes have been made
3. Write self-contained unit tests
    - Test inputs and outputs
    - Test the dataflow through the program
    - Test all the execution paths through the program
4. Stress-test the code; start simple and advance
5. Don’t implement new features if there are known bugs in the system
6. The target runtime environment is as important a design and implementation point as the purpose of the code. Design *and* test with that environment in mind
7. Test for portability: run code and tests on multiple machines/OSs
8. Before shipping code make sure that any debug/test modes are turned off

If you follow at least 50% of the tips in these notes you will write better code and it will have considerably fewer bugs.

Agenda

1. Types of tests
2. Unit test
3. Activity

Assert ends execution if a condition is false

assert_test.h

Include <asset.h>

Execution ends if assert evaluates as false

May not be best choice
Self driving car might not to exit while driving!

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <assert.h>
 4
 5 int main () {
 6     int n = 5;
 7     assert(n>0); //set assert(n>5) to see fail
 8
 9     int *p = &n;
10     assert(*p>0); //set assert(*p>5) to see fail
11
12     p = malloc(sizeof(int));
13     *p = 6;
14     assert(*p>0); //set assert(*p>6) to see fail
15
16
17     return 0;
18 }
```

Assert is helpful for sure, but somewhat limited
We build something a little more useful next

unittest.h provides some useful macros for unit testing

unittest.h

Standard header guards to prevent declaring multiple times

```
56 #ifndef __UNITTEST_H
57 #define __UNITTEST_H
58
59 // each test should start by setting the result count to zero
60 #define START_TEST_CASE(name) int _failures=0; char *_testname = (name);
61
62 // Check a condition; if false, print warning message.
63 // e.g., EXPECT(dict->start == NULL).
64 // note: the preprocessor
65 //   converts __LINE__ into line number where this macro was used, and
66 //   converts "#x" into a string constant for arg x.
67 #define EXPECT(x)                    \
68   if (!(x)) {                        \
69     _failures++;                     \
70     printf("Fail %s Line %d: [%s]\n", _testname, __LINE__, #x); \
71   }
72
73 // return the result count at the end of a test
74 #define END_TEST_CASE                          \
75   if (_failures == 0) {                        \
76     printf("PASS test %s\n\n", _testname);              \
77   } else {                              \
78     printf("FAIL test %s with %d errors\n\n", _testname, _failures); \
79   }
80
81 #define TEST_RESULT (_failures)
82
83 #endif // _UNITTEST_H
```

Recall that #define is expanded by the preprocessor

When preprocessor encounters START_TEST_CASE (name), it expands to int __failures …

unittest.h provides some useful macros for unit testing

unittest.h

```
56 #ifndef __UNITTEST_H
57 #define __UNITTEST_H
58
59 // each test should start by setting the result count to zero
60 #define START_TEST_CASE(name) int _failures=0; char *_testname = (name);
61
62 // Check a condition, if false, print warning message.
63 // e.g., EXPECT(dict->start == NULL).
64 // note: the preprocessor
65 //   converts __LINE__ into line number where this macro was used, and
66 //   converts "#x" into a string constant for arg x.
67 #define EXPECT(x)                                \
68   if (!(x)) {                                    \
69     _failures++;                                 \
70     printf("Fail %s Line %d: [%s]\n", _testname, __LINE__, #x); \
71   }
72
73 // return the result count at the end of a test
74 #define END_TEST_CASE                            \
75   if (_failures == 0) {                          \
76     printf("PASS test %s\n\n", _testname);              \
77   } else {                                       \
78     printf("FAIL test %s with %d errors\n\n", _testname, _failures); \
79   }
80
81 #define TEST_RESULT (_failures)
82
83 #endif // _UNITTEST_H
```

EXPECT takes a parameter such as "n > 10"
If that parameter evaluates to false, increment __failures and print error message

Built-in __LINE__ gives source code line number handy!

Converts #x into string representation of parameter (e.g., "n > 10")

unittest.h provides some useful macros for unit testing

unittest.h

```
56 #ifndef __UNITTEST_H
57 #define __UNITTEST_H
58
59 // each test should start by setting the result count to zero
60 #define START_TEST_CASE(name) int _failures=0; char *_testname = (name);
61
62 // Check a condition; if false, print warning message.
63 // e.g., EXPECT(dict->start == NULL).
64 // note: the preprocessor
65 //   converts __LINE__ into line number where this macro was used, and
66 //   converts "#x" into a string constant for arg x.
67 #define EXPECT(x)                    \
68   if (!(x)) {                        \
69     _failures++;                     \
70     printf("Fail %s Line %d: [%s]\n", _testname, __LINE__, #x); \
71   }
72
73 // return the result count at the end of a test
74 #define END_TEST_CASE                          \
75   if (_failures == 0) {                        \
76     printf("PASS test %s\n\n", _testname);           \
77   } else {                                     \
78     printf("FAIL test %s with %d errors\n\n", _testname, _failures); \
79   }
80
81 #define TEST_RESULT (_failures)
82
83 #endif // _UNITTEST_H
```

At END_TEST_CASE, if no errors, print PASS, else FAIL

TEST_RESULT returns number of failures

We can use macros in unittest.h to test the tree code from the lecture extra

treeA/tree.c

```
/***************** local types ****************/
typedef struct treenode {
  char *key;                  // search key for this item
  void *data;                 // pointer to data for this item
  struct treenode *left, *right;   // children
} treenode_t;

/***************** global types ****************/
typedef struct tree {
  struct treenode *root;                    // root of the tree
} tree_t;

/***************** global functions ****************/
/* that is, visible outside this file */
/* see tree.h for comments about exported functions */

/***************** local functions ****************/
/* not visible outside this file */
static treenode_t *tree_insert_helper(treenode_t *node,
                  const char *key, void *data);
static treenode_t *treenode_new(const char *key, void *data);
static void *tree_find_helper(treenode_t *node, const char *key);
static void tree_print_helper(tree_t *tree, treenode_t *node, int depth,
               FILE *fp,
               void (*itemprint)(FILE *fp, const char *key, void *data) );

static void tree_delete_helper(tree_t *tree, treenode_t *node,
                void (*itemdelete)(void *data) );

```

Review: tree nodes have:

- **Key**
- **Data**
- **Left and right**

Tree struct holds pointer to root

We can use macros in unittest.h to test the tree code from the lecture extra

treeA/tree.c

```c
#ifdef UNIT_TEST
#include "unittest.h"

////////////////////////////////////
// create and validate an empty tree
int test_newtree0() {
  START_TEST_CASE("newtree0");
  tree_t *tree = tree_new();
  EXPECT(tree != NULL);
  EXPECT(tree->root == NULL);

  EXPECT(tree_find(tree, "hello") == NULL);

  tree_delete(tree, NULL);
  EXPECT(count_net() == 0);

  END_TEST_CASE;
  return TEST_RESULT;
}

int main(const int argc, const char *argv[]) {
  int failed = 0;

  failed += test_newtree0();

  if (failed) {
    printf("FAIL %d test cases\n", failed);
    return failed;
  } else {
    printf("PASS all test cases\n");
    return 0;
  }
}

#endif // UNIT_TEST
```

If compiled with –DUNIT_TEST include this code, otherwise do not

Give test a name

```c
 59 // each test should start by setting the result
count to zero
 60 #define START_TEST_CASE(name) int _failures=0;
char *_testname = (name);
 61
```

Macro creates inf failures=0 and string for test name

Main function added if compiled with unit test, otherwise there would be not main() for this .c file

Run unit test in function `test_newtree0()`

We can use macros in unittest.h to test the tree code from the lecture extra

treeA/tree.c

```
#ifdef UNIT_TEST
#include "unittest.h"

////////////////////////////////////
// create and validate an empty tree
int test_newtree0() {
  START_TEST_CASE("newtree0");
  tree_t *tree = tree_new();
  EXPECT(tree != NULL);
  EXPECT(tree->root == NULL);

  EXPECT(tree_find(tree, "hello") == NULL);

  tree_delete(tree, NULL);
  EXPECT(count_net() == 0);

  END_TEST_CASE;
  return TEST_RESULT;
}

int main(const int argc, const char *argv[]) {
  int failed = 0;

  failed += test_newtree0();

  if (failed) {
    printf("FAIL %d test cases\n", failed);
    return failed;
  } else {
    printf("PASS all test cases\n");
    return 0;
  }
}

#endif // UNIT_TEST
```

EXPECT macro tests if parameter is true

Keep track of number of __failures

```
67 #define EXPECT(x)                   \
68   if (!(x)) {                       \
69     _failures++;                    \
70     printf("Fail %s Line %d: [%s]\n", _testname, __LINE__, #x); \
71   }
72
```

Run a number of test

Return number of errors

```
74 #define END_TEST_CASE                       \
75   if (_failures == 0) {                      \
76     printf("PASS test %s\n\n", _testname);            \
77   } else {                                   \
78     printf("FAIL test %s with %d errors\n\n", _testname,
_failures); \
79   }
80
81 #define TEST_RESULT (_failures)
```

Output results of tests

Compile with –DUNIT_TEST to have the pre-processor include this code

treeA/tree.c

```
#ifdef UNIT_TEST
#include "unittest.h"

////////////////////////////////////
// create and validate an empty tree
int test_newtree0() {
  START_TEST_CASE("newtree0");
  tree_t *tree = tree_new();
  EXPECT(tree != NULL);
  EXPECT(tree->root == NULL);

  EXPECT(tree_find(tree, "hello") == NULL);

  tree_delete(tree, NULL);
  EXPECT(count_net() == 0);

  END_TEST_CASE;
  return TEST_RESULT;
}

int main(const int argc, const char *argv[]) {
  int failed = 0;

  failed += test_newtree0();

  if (failed) {
    printf("FAIL %d test cases\n", failed);
    return failed;
  } else {
    printf("PASS all test cases\n");
    return 0;
  }
}

#endif // UNIT_TEST
```

```
$ make unittest
$ ./unittest
PASS test newtree0

PASS all test cases
```

Remember: because of #ifdef UNIT_TEST, test code and main() only included if flag is set at compile time

```
27 unittest: tree.h tree.c memory.h unittest.h memory.o
28     $(CC) $(CFLAGS) -DUNIT_TEST tree.c memory.o -o unittest
```

If UNIT_TEST flag not set, preprocessor does not include test code for compilation

Compile with –E flag to see pre-processor's output expanding macros

-E flag shows pre-processor's output expanding macro

treeA/tree.c

```
$ mygcc –DUNIT_TEST –E tree.c memory.c
```

```
#ifdef UNIT_TEST
#include "unittest.h"

////////////////////////////////////
// create and validate an empty tree
int test_newtree0() {
  START_TEST_CASE("newtree0");
  tree_t *tree = tree_new();
  EXPECT(tree != NULL);
  EXPECT(tree->root == NULL);
```

```
60 #define START_TEST_CASE(name) int _failures=0; char
*_testname = (name);
```

```
  tree_delete(tree, NULL);
  EXPECT(count_net() == 0);

  END_TEST_CASE;
  return TEST_RESULT;
}

int main(const int argc, const char *argv[]) {
  int failed = 0;

  failed += test_newtree0();

  if (failed) {
    printf("FAIL %d test cases\n", failed);
    return failed;
  } else {
    printf("PASS all test cases\n");
    return 0;
  }
}

#endif // UNIT_TEST
```

START_TEST_CASE("newtree0")

```
# 1 "unittest.h" 1
# 238 "tree.c" 2

int test_newtree0()
{
  int _failures=0; char *_testname = ("newtree0");;
  tree_t *tree = tree_new();
  if (!(tree !=
# 245 "tree.c" 3 4
 ((void *)0)
# 245 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 245, "tree
!= NULL"); };
  if (!(tree->root ==
# 246 "tree.c" 3 4
 ((void *)0)
# 246 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 246,
"tree->root == NULL"); };

  if (!(tree_find(tree, "hello") ==
# 248 "tree.c" 3 4
 ((void *)0)
# 248 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 248,
"tree_find(tree, \"hello\") == NULL"); };

  tree_delete(tree,
# 250 "tree.c" 3 4
                   ((void *)0)
# 250 "tree.c"
                       );
  if (!(count_net() == 0)) { _failures++; printf("Fail %s Line %d:
[%s]\n", _testname, 251, "count_net() == 0"); };

  if (_failures == 0) { printf("PASS test %s\n\n", _testname); } else {
printf("FAIL test %s with %d errors\n\n", _testname, _failures); };
  return (_failures);
}
```

Compile with –E flag to see pre-processor's output expanding macros

-E flag shows pre-processor's output expanding macro

treeA/tree.c

```
$ mygcc –DUNIT_TEST –E tree.c memory.c
```

```
#ifdef UNIT_TEST
#include "unittest.h"

////////////////////////////////////
// create and validate an empty tree
int test_newtree0() {
  START_TEST_CASE("newtree0");
  tree_t *tree = tree_new();
  EXPECT(tree != NULL);
  EXPECT(tree->root == NULL);
```

```
 67 #define EXPECT(x)                    \
 68   if (!(x)) {                        \
 69     _failures++;                     \
 70     printf("Fail %s Line %d: [%s]\n", _testname,
__LINE__, #x); \
 71   }
 72
```

```
  return TEST_RESULT;
}

int main(const int argc, const char *argv[]) {
  int failed = 0;

  failed += test_newtree0();

  if (failed) {
    printf("FAIL %d test cases\n", failed);
    return failed;
  } else {
    printf("PASS all test cases\n");
    return 0;
  }
}

#endif // UNIT_TEST
```

```
# 1 "unittest.h" 1
# 238 "tree.c" 2

int test_newtree0()
{
  int _failures=0; char *_testname = ("newtree0");;
  tree_t *tree = tree_new();
  if (!(tree !=
# 245 "tree.c" 3 4
 ((void *)0)
# 245 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 245, "tree
!= NULL"); };
  if (!(tree->root ==
# 246 "tree.c" 3 4
 ((void *)0)
# 246 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 246,
"tree->root == NULL"); };

  if (!(tree_find(tree, "hello") ==
# 248 "tree.c" 3 4
 ((void *)0)
# 248 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 248,
"tree_find(tree, \"hello\") == NULL"); };

  tree_delete(tree,
# 250 "tree.c" 3 4
                   ((void *)0)
# 250 "tree.c"
                       );
  if (!(count_net() == 0)) { _failures++; printf("Fail %s Line %d:
[%s]\n", _testname, 251, "count_net() == 0"); };

  if (_failures == 0) { printf("PASS test %s\n\n", _testname); } else {
printf("FAIL test %s with %d errors\n\n", _testname, _failures); };
  return (_failures);
}
```

EXPECT(tree != NULL)

Note: NULL also expands to (void *)0

Compile with –E flag to see pre-processor's output expanding macros

-E flag shows pre-processor's output expanding macro

treeA/tree.c

```
$ mygcc –DUNIT_TEST –E tree.c memory.c
```

```
#ifdef UNIT_TEST
#include "unittest.h"

////////////////////////////////////
// create and validate an empty tree
int test_newtree0() {
  START_TEST_CASE("newtree0");
  tree_t *tree = tree_new();
  EXPECT(tree != NULL);
  EXPECT(tree->root == NULL);
```

```
 67 #define EXPECT(x)                    \
 68   if (!(x)) {                        \
 69     _failures++;                     \
 70     printf("Fail %s Line %d: [%s]\n", _testname,
__LINE__, #x); \
 71   }
 72
```

```
  return TEST_RESULT;
}

int main(const int argc, const char *argv[]) {
  int failed = 0;

  failed += test_newtree0();

  if (failed) {
    printf("FAIL %d test cases\n", failed);
    return failed;
  } else {
    printf("PASS all test cases\n");
    return 0;
  }
}

#endif // UNIT_TEST
```

EXPECT(tree->root = NULL)

Note: NULL also expands to (void *)0

```
# 1 "unittest.h" 1
# 238 "tree.c" 2

int test_newtree0()
{
  int _failures=0; char *_testname = ("newtree0");;
  tree_t *tree = tree_new();;
  if (!(tree !=
# 245 "tree.c" 3 4
 ((void *)0)
# 245 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 245, "tree
!= NULL"); };
  if (!(tree->root ==
# 246 "tree.c" 3 4
 ((void *)0)
# 246 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 246,
"tree->root == NULL"); };

  if (!(tree_find(tree, "hello") ==
# 248 "tree.c" 3 4
 ((void *)0)
# 248 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 248,
"tree_find(tree, \"hello\") == NULL"); };

  tree_delete(tree,
# 250 "tree.c" 3 4
                   ((void *)0)
# 250 "tree.c"
                       );
  if (!(count_net() == 0)) { _failures++; printf("Fail %s Line %d:
[%s]\n", _testname, 251, "count_net() == 0"); };

  if (_failures == 0) { printf("PASS test %s\n\n", _testname); } else {
printf("FAIL test %s with %d errors\n\n", _testname, _failures); };
  return (_failures);
}
```

Compile with –E flag to see pre-processor's output expanding macros

```
#ifdef UNIT_TEST
#include "unittest.h"

////////////////////////////////////
// create and validate an empty tree
int test_newtree0() {
  START_TEST_CASE("newtree0");
  tree_t *tree = tree_new();
  EXPECT(tree != NULL);
  EXPECT(tree->root == NULL);

  EXPECT(tree_find(tree, "hello") == NULL);

  tree_delete(tree, NULL);
  EXPECT(count_net() == 0);

  END_TEST_CASE;
  return TEST_RESULT;
}

int main(const int argc, const char *argv[]) {
  int failed = 0;

  failed += test_newtree0();

  if (failed) {
```

-E flag shows pre-processor's output expanding macro

treeA/tree.c

```
$ mygcc –DUNIT_TEST –E tree.c memory.c
```

```
# 1 "unittest.h" 1
# 238 "tree.c" 2

int test_newtree0()
{
  int _failures=0, char *_testname = ("newtree0");;
  tree_t *tree = tree_new();
  if (!(tree !=
# 245 "tree.c" 3 4
 ((void *)0)
# 245 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 245, "tree
!= NULL"); };
  if (!(tree->root ==
# 246 "tree.c" 3 4
 ((void *)0)
# 246 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 246,
"tree->root == NULL"); };

  if (!(tree_find(tree, "hello") ==
# 248 "tree.c" 3 4
 ((void *)0)
# 248 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 248,
"tree_find(tree, \"hello\") == NULL"); };

  tree_delete(tree,
# 250 "tree.c" 3 4
                   ((void *)0)
# 250 "tree.c"
                        );
  if (!(count_net() == 0)) { _failures++; printf("Fail %s Line %d:
[%s]\n", _testname, 251, "count_net() == 0"); };

  if (_failures == 0) { printf("PASS test %s\n\n", _testname); } else {
printf("FAIL test %s with %d errors\n\n", _testname, _failures); };
  return (_failures);
}
```

END_TEST_CASE adds print statements

```
74 #define END_TEST_CASE                       \
 75   if (_failures == 0) {                     \
 76     printf("PASS test %s\n\n",
_testname);            \
 77   } else {                          \
 78     printf("FAIL test %s with %d errors\n\n",
_testname, _failures); \
 79   }
 80
 81 #define TEST_RESULT (_failures)
```

Compile with –E flag to see pre-processor's output expanding macros

```
#ifdef UNIT_TEST
#include "unittest.h"

////////////////////////////////////
// create and validate an empty tree
int test_newtree0() {
  START_TEST_CASE("newtree0");
  tree_t *tree = tree_new();
  EXPECT(tree != NULL);
  EXPECT(tree->root == NULL);

  EXPECT(tree_find(tree, "hello") == NULL);

  tree_delete(tree, NULL);
  EXPECT(count_net() == 0);

  END_TEST_CASE;
  return TEST_RESULT;
}

int main(const int argc, const char *argv[]) {
  int failed = 0;

  failed += test_newtree0();

  if (failed) {
```

-E flag shows pre-processor's output expanding macro

treeA/tree.c

```
$ mygcc –DUNIT_TEST –E tree.c memory.c
```

TEST_RESULT returns __failure count

```
# 1 "unittest.h" 1
# 238 "tree.c" 2

int test_newtree0()
{
  int _failures=0; char *_testname = ("newtree0");
  tree_t *tree = tree_new();
  if (!(tree !=
# 245 "tree.c" 3 4
 ((void *)0)
# 245 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 245, "tree
!= NULL"); };
  if (!(tree->root ==
# 246 "tree.c" 3 4
 ((void *)0)
# 246 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 246,
"tree->root == NULL"); };

  if (!(tree_find(tree, "hello") ==
# 248 "tree.c" 3 4
 ((void *)0)
# 248 "tree.c"
 )) { _failures++; printf("Fail %s Line %d: [%s]\n", _testname, 248,
"tree_find(tree, \"hello\") == NULL"); };

  tree_delete(tree,
# 250 "tree.c" 3 4
                   ((void *)0)
# 250 "tree.c"
                        );
  if (!(count_net() == 0)) { _failures++; printf("Fail %s Line %d:
[%s]\n", _testname, 251, "count_net() == 0"); };

  if (_failures == 0) { printf("PASS test %s\n\n", _testname); } else {
printf("FAIL test %s with %d errors\n\n", _testname, _failures); };
  return (_failures);
}
```

```
74 #define END_TEST_CASE                          \
 75   if (_failures == 0) {                       \
 76     printf("PASS test %s\n\n",
_testname);            \
 77   } else {                          \
 78     printf("FAIL test %s with %d errors\n\n",
_testname, _failures); \
 79   }
 80
 81 #define TEST_RESULT (_failures)
```

Agenda

1. Types of tests
2. Unit test
3. Activity