

Week 11 Lecture 1

Unit Testing

Unit Testing

Test Driven Development

- 1) Write a test
- 2) Run all tests to make sure the old ones pass and the new one fails
- 3) Write the smallest program that make the test pass
- 4) Run all test to make sure that all tests pass
- 5) Refactor

CUnit

- Problem: Running all tests can take a long time.
- Solution: Write a program to run the tests.
- CUnit: A program to make writing tests easier.

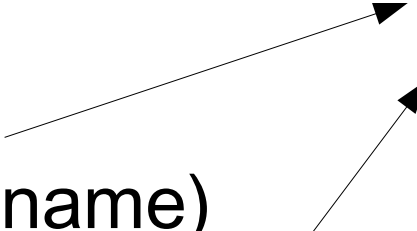
Example: Hello World

- Unit tests are meant to test units, so we need to refactor `hello.c` into a unit and a main function
- We can then test the individual functions.

hello.h

- The header file declares two functions

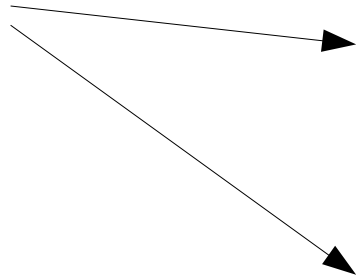
- getName
- sayHello(name)



```
int getName(char *name);  
int sayHello(char *name);  
hello.h (END)
```

hello.c

- Definitions



```
#include <stdio.h>

int getName(char *name) {
    int returnVal;
    printf("Enter your name> ");
    returnVal = scanf("%10s", name);
    printf("returning %d characters\n", returnVal);
    return returnVal;
}

int sayHello(char *name) {
    return printf("Hello %s\n", name);
}

hello.c (END)
```

main.c

- Declarations
- Execution
- The intention of the main function is clearer

```
#include <stdio.h>
#include "hello.h"

int main(int argc, char *argv[]) {
    char name[11];

    getName(name);
    sayHello(name);
    return 0;
}
main.c (END)
```


Makefile

- Variable used for compile flags
- App depends on main and hello
- New: test creates test_hello and runs it

```
CFLAGS = -Wall -std=c11
all: hello

hello: hello.o main.o
    gcc -o hello hello.o main.o

hello.o: hello.c
    gcc -c $(CFLAGS) hello.c

main.o: main.c
    gcc -c $(CFLAGS) main.c

clean:
    rm -f hello test_hello *.o test/*.o

test: hello.o
    gcc $(CFLAGS) -o test_hello test/hello_test.c hello.o -lcunit
    ./hello_test

Makefile (END)
```

Test Functions

- Test getName
 - Create variable
 - Prompt for input
 - Call function under test
 - Assert got input
- Test sayHello
 - Check that ten letters printed
 - Functions return values:
printf returns number of
chars printed.

```
/****** Test case functions *****/  
  
void getName_test(void) {  
    char name[11];  
    printf("\nPlease enter \"Nat\\n");  
    getName(name);  
    CU_ASSERT_STRING_EQUAL(name, "Nat");  
}  
  
void sayHello_test(void) {  
    CU_ASSERT_EQUAL(sayHello("Nat"), 10);  
}
```

Run tests

- Create a registry
- Create a suite
- Add tests to suite
 - Suite
 - Test message
 - Test function

```
/* ***** Test Runner Code goes here ***** */  
  
int main ( void )  
{  
    CU_pSuite pSuite = NULL;  
  
    /* initialize the CUnit test registry */  
    if ( CUE_SUCCESS != CU_initialize_registry() )  
        return CU_get_error();  
  
    /* add a suite to the registry */  
    pSuite = CU_add_suite( "hello_test_suite", init_suite, clean_suite );  
    if ( NULL == pSuite ) {  
        CU_cleanup_registry();  
        return CU_get_error();  
    }  
  
    /* add the tests to the suite */  
    if ( (NULL == CU_add_test(pSuite, "getName Test", getName_test)) ||  
         (NULL == CU_add_test(pSuite, "sayHello Test", sayHello_test)) ) {  
        CU_cleanup_registry();  
        return CU_get_error();  
    }  
  
    // Run all tests using the basic interface  
    CU_basic_set_mode(CU_BRM_VERBOSE);  
    CU_basic_run_tests();  
    printf("\n");  
    CU_basic_show_failures(CU_get_failure_list());  
    printf("\n\n");  
  
    /* Clean up registry and return */  
    return CU_get_error();  
}
```

CUnit Overview

- CUnit is a program for unit testing.
- It comprises a testing framework and a test runner: both are C programs.
 - The testing framework supplies asserts to build tests.
 - The test runner calls the tests

Tests in CUnit

Assertions in CUnit

- CU_FAIL always fails; CU_PASS always passes.

```
/*  
 * Testing cunit  
 * Failure expected on last test.  
 */  
void test_cunit(void) {  
    CU_ASSERT(1 == 1);  
    CU_ASSERT_TRUE(1 == 1);  
    CU_ASSERT_FALSE(1 != 1);  
    CU_ASSERT_EQUAL(1, 1);  
    CU_ASSERT_NOT_EQUAL(1, 2);  
    CU_PASS("CU_PASS");  
    CU_FAIL("CU_FAIL");  
}
```

CUnit tests are C functions

- Void functions with no parameters
- `valid_operator`.
 - Checks all characters
 - Valid operators return true
 - Invalid operators return false

```
void test_valid_operator(void) {  
    for (char op = ' '; op <= '~'; op++) {  
        if ('x' == op  
            || '+' == op  
            || '-' == op  
            || '*' == op  
            || '/' == op  
            || '!' == op  
            || 'i' == op  
            || 'b' == op  
            || 'g' == op  
            || 'p' == op  
            || 's' == op  
            || 'm' == op  
        ) {  
            CU_ASSERT_TRUE(valid_operator(op));  
        } else {  
            CU_ASSERT_FALSE(valid_operator(op));  
        }  
    }  
}
```

Floats and Precision

- Floating point equality is only approximate
- Third parameter is precision
 - They are equal if they are equal within precision

```
void test_divide(void) {  
    CU_ASSERT_DOUBLE_EQUAL(divide(1, 2), 0.5, .001);  
    CU_ASSERT_DOUBLE_EQUAL(divide(-1, 2), -0.5, .001);  
    CU_ASSERT_DOUBLE_EQUAL(divide(1, -2), -0.5, .001);  
    CU_ASSERT_DOUBLE_EQUAL(divide(-1, -2), 0.5, .001);  
    CU_ASSERT_DOUBLE_EQUAL(divide(1.1, 2.1), 0.5238, .001);  
    CU_ASSERT_DOUBLE_EQUAL(divide(-1.1, 2.2), (-1.1/2.2), .001);  
    CU_ASSERT_DOUBLE_EQUAL(divide(1.1, -2.2), (1.1/-2.2), .001);  
    CU_ASSERT_DOUBLE_EQUAL(divide(-1.1, -2.2), (-1.1/-2.2), .001);  
}
```


CUnit Testing Framework

Testing Framework Runs Tests

- The command `make test` compiles the test and testing framework
 - It is also compiled with `calc.h` and `calc.c`
 - These files that define the module
 - The tests are run against the module
- Running `./calc_test` runs the program compiled from the tests and the framework.

Add Tests to Testing Framework

- You need to tell the testing framework which tests to run.
- In CUnit there are three steps:
 - 1)Set up the test registry
 - 2)Create a suite
 - 3)Add tests to suite

Setting up CUnit Tests

```
int main ( void )
{
    CU_pSuite pSuite = NULL;

    /* initialize the CUnit test registry */
    if ( CUE_SUCCESS != CU_initialize_registry() ) {
        fprintf(stderr, "Failed to initialize the CUnit test registry\n");
        return CU_get_error();
    }

    /* add a suite to the registry */
    pSuite = CU_add_suite( "calc test suite", init_suite, clean_suite );
    if ( NULL == pSuite ) {
        fprintf(stderr, "Failed to add suite to registry\n");
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* add the tests to the suite */
    if ( !add_tests_to_suit(pSuite) ) {
        fprintf(stderr, "Failed to add tests to suite\n");
        fprintf(stderr, "\t%s\n", CU_get_error_msg());
        CU_cleanup_registry();
        return CU_get_error();
    }
}
```

Create the registry

- Call: CU_initialize_registry
 - In the segment below we also check that it works and print an error message and quit if it does not.
 - There is only one CUnit registry per test runner.

```
/* initialize the CUnit test registry */  
if ( CUE_SUCCESS != CU_initialize_registry() ) {  
    fprintf(stderr, "Failed to initialize the CUnit test registry\n");  
    return CU_get_error();  
}
```

Create a suite

- Call: `CU_add_suite` storing returned value in a `CU_pSuite` variable
 - Returns `NULL` on failure.

```
int main ( void )
{
    CU_pSuite pSuite = NULL;
```

```
/* add a suite to the registry */
pSuite = CU_add_suite( "calc test suite", init_suite, clean_suite );
if ( NULL == pSuite ) {
    fprintf(stderr, "Failed to add suite to registry\n");
    CU_cleanup_registry();
    return CU_get_error();
}
```

Function defined to add tests

- Function defined in test.c
 - Called in test_main.c
- Returns true if every **CU_add_test** returns true
- Allows tests to be added to suite in test.c

```
int add_tests_to_suit(CU_pSuite pSuite) {  
    return  
    (  
        NULL != CU_add_test(pSuite, "Testing: divide(double, double)",  
                             test_divide) &&  
        NULL != CU_add_test(pSuite, "Testing: valid_operator(char)",  
                             test_valid_operator) &&  
        NULL != CU_add_test(pSuite, "CUnit Test", test_cunit)  
    );  
}
```

Function to add test called

- Call `add_tests_to_suite`
 - Pass in `CU_pSuite` created in step 2.
 - Print error, cleanup and return error if function fails.

```
if ( !add_tests_to_suit(pSuite) ) {  
    fprintf(stderr, "Failed to add tests to suite\n");  
    fprintf(stderr, "\t%s\n", CU_get_error_msg());  
    CU_cleanup_registry();  
    return CU_get_error();  
}
```


Running the tests

- After creating the registry, creating a suite, and adding tests to suite, we need to run the tests.
 - `CU_basic_run_tests()` will run the tests.

```
printf("\n\n***** Running Tests *****\n");
CU_basic_set_mode(CU_BRM_VERBOSE);
CU_basic_run_tests();
printf("\n***** Tests Run *****\n");
printf("\n***** Failed Tests *****\n");
CU_basic_show_failures(CU_get_failure_list());
printf("\n***** Failed Tests *****\n\n");
```

Seeing failures

- We want to see failed tests.
 - CU_basic_set_mode: info to display
 - CU_BRM_VERBOSE: max info
 - CU_basic_show_failures: shows failures
 - CU_get_failure_list: gets failures

```
printf("\n\n***** Running Tests *****\n");
CU_basic_set_mode(CU_BRM_VERBOSE);
CU_basic_run_tests();
printf("\n***** Tests Run *****\n");
printf("\n***** Failed Tests *****\n");
CU_basic_show_failures(CU_get_failure_list());
printf("\n***** Failed Tests *****\n\n");
```