Unit Testing con CUnit

Introducción

¿Qué es CUnit?

Es una biblioteca que permite realizar testeos unitarios a programas escritos en C. Hacer esto es una buena práctica para tener una mayor sensación de la robustez del código, y asegurar al menos que lo que está testeado, anda.

Además, al definir las pruebas como **test cases**, se pueden repetir de 1 a N veces, asegurándote de que tu programa no rompa aleatoriamente o que no se rompió después de un cambio grande.

¿Y eso de qué me sirve para el TP?

Ponele que estás a 6 horas de la entrega, mucho café, tenés que arreglar un bug pero no querés tocar mucho por miedo a romper las demás funcionalidades. "Nah, mejor lo dejo así, ojalá no rompa" ← Nunca digas eso. Jamás de los jamases. Nun-ca. Abrí el eclipse (o el vi, si sos macho) y ponete a revisar el código 😃

Ok, lo arreglaste, tuviste que cambiar el core del tp... ¿cómo sabés que sigue andando todo? Fácil: corré los test. No hace falta volverte loco revisando logs de 300 líneas, o llenar el debugger de breakpoints. Si tenés un buen conjunto de pruebas, solo tenés que volverlas a correr y ver que pasan luego de hacer los cambios.

Ok, compro, ¡quiero 8! ¿Dónde lo bajo?

En las VMs de la cátedra ya viene instalado.

Si estás bajo tu propio linux, en Ubuntu es:

```bash
sudo apt-get install libcunit1 libcunit1-doc libcunit1-dev
```

¿Cómo funciona?

Antes que nada, vamos a definir algunos conceptos:

- **Test case**: Es una función destinada a probar una funcionalidad. Idealmente se compone de 3 partes:
  - *Inicialización*: generar el fixture de datos que serán utilizados durante la prueba
  - *Lógica*: llamar a las funciones que hagan falta
  - *Aserciones*: validar que los resultados sean los que nosotros esperamos

Ejemplo de un test case en CUnit:

```c
void test_strlen_devuelve_la_longitud_del_string() {
    char* unaCadena = "Empanada";
    int longitud = strlen(unaCadena);
    CU_ASSERT_EQUAL(longitud, 8);
}
```

- **Suite**: Es un conjunto de test cases que prueban una funcionalidad común. Opcionalmente además puede tener funciones de inicialización y limpieza. Estas funciones se ejecutan *una sola vez* antes y después de correr el suite respectivamente.
- **Setup**: Función que se ejecuta una vez *antes* de cada test case del suite. Sirve para inicializar un fixture de datos común, crear archivos, etc.

Al correr un proyecto de CUnit, se corren los test cases de todas las suites agregadas, y se muestra un resumen en consola indicando:

- Cuántos test pasaron
- Cuántos fallaron (y cuáles)

Si en el test del ejemplo anterior cambiaramos el 8 por un 4, el output de consola sería:

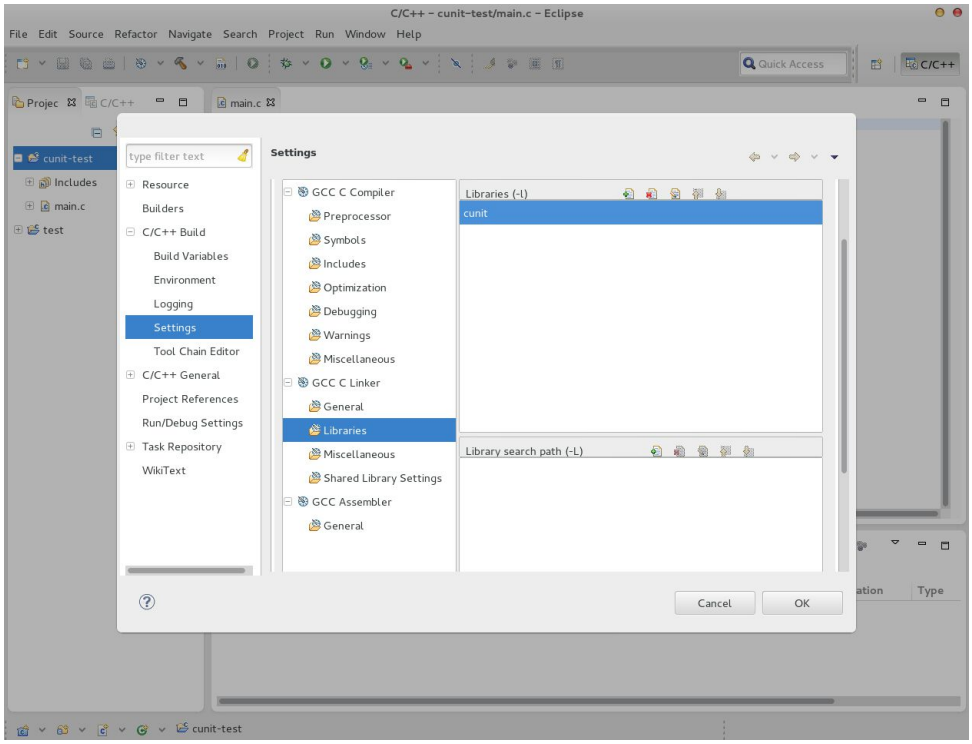
```
     CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/


Suite: Suite de prueba
  Test: strlen devuelve la longitud del string ...FAILED
    1. src/example.c:6  - CU_ASSERT_EQUAL(longitud,4)

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      0      1        0
             asserts      1      1      0      1      n/a

Elapsed time =    0.000 seconds
```

Configurando el proyecto de Eclipse

Vamos al eclipse, creamos el proyecto con el nombre que nos guste. Yo le puse `cunit-test`, pero ponele el que vos quieras. Para que todo funcione tenemos que linkear con la biblioteca. ⇒ Hagamos eso:

Botón derecho en el proyecto → `Properties` → `C/C++ Build` → `Settings` → `GCC C Linker` → `Libraries`

... y agregamos la biblioteca `cunit` en `Libraries`.



Ejemplo 1

Ahora que ya agregamos la biblioteca, vamos a crear un `main.c`, que incluya al header de CUnit.

La biblioteca tiene 4 formas de mostrar la información:

- **Automatic**: Guarda el output en un archivo xml.
- **Basic**: Muestra el output por la salida estándar (la terminal).
- **Console**: Provee interacción con el usuario, y te deja navegar por menús, activar o desactivar los suites que queremos correr, etc.
- **Curses**: Similar al anterior pero con una interfaz más linda.

Por ahora vamos a usar el Basic, así que incluímos `CUnit/Basic.h`.

```c
#include <CUnit/Basic.h>
```

Ahora definimos la función `main()` así:

```c
int main() {
    CU_initialize_registry();

    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();

    return CU_get_error();
}
```

¿Qué hace cada función?

- `CU_initialize_registry()` inicializa un registro de suites vacío.
- `CU_basic_set_mode(CU_BRM_VERBOSE)` setea la biblioteca de modo tal que muestre la mayor cantidad de información posible (con el flag `CU_BRM_VERBOSE` [1])
- `CU_basic_run_tests()` corre los tests.
- `CU_cleanup_registry()` destruye todas las estructuras y los tests creados, libera la memoria utilizada por la biblioteca (para evitar memory leaks). CUnit nos pide que lo llamemos y que sea lo último que hagamos.
- `CU_get_error()` devuelve un código al finalizar la ejecución, que puede ser de error o de ejecución correcta[2].

Claramente nada de esto sirve si no armamos un suite (o sea, ¡si no tenemos casos de prueba!).

Es importante destacar que CUnit nos pide respetar un contrato y es que los tests son funciones que no devuelven valores ni tampoco reciben parámetros. Por ende, deben definirse de esta manera: `void miTest(void)`

Los tests podrían ser algo así:

```c
void test1() {
    printf("Soy el test 1!, y pruebo que 2 sea igual a 1+1\n");
    CU_ASSERT_EQUAL(1+1, 2);
}

void test2() {
    printf("Soy el test 2!, y doy segmentation fault\n");
    char *ptr = NULL;
    *ptr = 9;
}

void test3() {
    printf("Soy el test 3!\n");
}
```

Solamente en el primer test hacemos un assert para chequear que todo lo que hicimos en ese bloque de código haya salido como esperamos. Por cada test podemos hacer tantos

asserts como queramos. Incluso podemos no hacerlos (como en el `test2()` y `test3()`), todo depende de lo que queremos testear.

Ahora que tenemos los tests, sólo queda meterlos dentro un suite. ¿Cómo lo hacemos? Estos tests hay que meterlos dentro de un suite, así que *luego de la primera línea del main* agregamos:

```c
int main() {
    CU_initialize_registry();

    CU_pSuite prueba = CU_add_suite("Suite de prueba", NULL, NULL);
    CU_add_test(prueba, "uno", test1);
    CU_add_test(prueba, "dos", test2);
    CU_add_test(prueba, "tres", test3);

    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();

    return CU_get_error();
}
```

¿Qué hace cada función?

- `CU_add_suite(strName, pInit, pClean)` crea el suite donde vamos a meter nuestros tests.
  - `strName` es el nombre con el que se va a mostrar nuestra suite.
  - `pInit` es un puntero a la función de inicialización que se invocará antes de correr la suite.
  - `pClean` es un puntero a la función de limpieza que se invocará después de correr la suite.

  En ambos casos podemos pasarle NULL si no queremos usar ninguna función hecha por nosotros.

  La función retorna un puntero a la suite creada (o `NULL` en caso de error).
- `CU_add_test(pSuite, strName, pTest)` agrega un test a una suite.
  - `pSuite` es un puntero a la suite donde agregaremos el test (el que nos devuelve `CU_add_suite()`).
  - `strName` es un nombre que le damos al caso de prueba.
  - `pTest` es un puntero a la función que es el test en sí.

Ya tenemos un set de pruebas y configuramos todo. → Run!

```
     CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/


Suite: Suite de prueba
  Test: uno ...Soy el test 1!, y pruebo que 2 sea igual a 1+1
passed
  Test: dos ...Soy el test 2!, y doy segmentation fault
[1]    15177 segmentation fault (core dumped)  ./bin/tests.out
```

Como se puede ver el *test1* pasó ya que el assert se cumplió, el segundo rompió y por tanto *el test3 ni se llegó a ejecutar*.

Si sacás el test2 del suite, podés ver cómo los dos que quedaron pasan, y el programa finaliza correctamente.

El test3 no prueba nada, ya que no tiene ningún assert, así que por definición pasa.

Ejemplo 2: Lector de archivos

Supongamos que hicimos un par de funciones de manejo de archivos de texto y las queremos testear.

El código a probar es el siguiente, es una función que cuenta el número de ocurrencias de un carácter en un archivo:

lector.h lector.c

```c
#ifndef LECTOR_H_
#define LECTOR_H_

int archivo_contar(char* path, char c);

#endif
```

Si tuviéramos un archivo `algo.txt` que tiene de contenido "hola mundo", entonces `archivo_contar("algo.txt", 'o')` devolvería 2, ya que hay dos letras 'o' en la cadena. Eso es lo que queremos probar.

Ahora que ya tenemos el código, pasemos a la parte importante. Agreguemos al `main.c` nuestro header:

```c
#include "lector.h"
#include <CUnit/Basic.h>

int main() {
    CU_initialize_registry();

    CU_pSuite prueba = CU_add_suite("Suite de prueba", NULL, NULL);

    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();

    return CU_get_error();
}
```

Lo primero que estaría bueno probar es la primer parte, cuando el archivo que recibe como parámetro no existe. Armamos el test case para eso, lo agregamos al suite en el `main()` y lo corremos:

```c
#include "lector.h"
#include <CUnit/Basic.h>

void test_contar_devuelve_menos1_si_el_archivo_no_existe() {
    int cantidad = archivo_contar("askjd.txt", 'f');
    CU_ASSERT_EQUAL(cantidad,-1);
}

int main() {
    CU_initialize_registry();

    CU_pSuite prueba = CU_add_suite("Suite de prueba", NULL, NULL);
    CU_add_test(
        prueba,
        "archivo_contar devuelve -1 si el archivo no existe",
        test_contar_devuelve_menos1_si_el_archivo_no_existe
    );

    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();

    return CU_get_error();
}
```

```
Suite: Suite de prueba
  Test: archivo_contar devuelve -1 si el archivo no existe ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      1      0        0
             asserts      1      1      1      0      n/a
```

Veremos que pasa correctamente. Listo.

Ahora hay que probar el caso más común, que dado un archivo que exista, lo lea y devuelva la cantidad correcta de ocurrencias:

```c
// ...

void test_contar_devuelve_el_numero_exacto_de_ocurrencias() {
    char *path = "prueba.txt";
    FILE *archivo = fopen(path, "w+");
    fprintf(archivo, "aca hay como 4 as");
    fflush(archivo); //para que se guarde en disco ahora
    int cantidad = archivo_contar(path, 'a');
    CU_ASSERT_EQUAL(cantidad, 4);
}

int main() {
    CU_initialize_registry();

    CU_pSuite prueba = CU_add_suite("Suite de prueba", NULL, NULL);
    CU_add_test(
        prueba,
        "archivo_contar devuelve -1 si el archivo no existe",
        test_contar_devuelve_menos1_si_el_archivo_no_existe
    );
    CU_add_test(
        prueba,
        "archivo_contar devuelve el numero exacto de concurrencias",
        test_contar_devuelve_el_numero_exacto_de_ocurrencias
    );

    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();

    return CU_get_error();
}
```

¿Lo corremos?

```
Suite: Suite de prueba
  Test: archivo_contar devuelve -1 si el archivo no existe ...passed
  Test: archivo_contar devuelve el numero exacto de concurrencias ...FAILED
    1. src/example.c:15  - CU_ASSERT_EQUAL(cantidad,4)

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      2      2      1      1        0
             asserts      2      2      1      1      n/a
```

Nos dice que falló y en qué assert. A ver qué pasó...

Ah, estábamos recorriendo mal el array, desde 1 en lugar de desde 0:

@code{29-43} c{8}

Lo cambiamos y vemos que ahora anda.

```
Suite: Suite de prueba
  Test: archivo_contar devuelve -1 si el archivo no existe ...passed
  Test: archivo_contar devuelve el numero exacto de concurrencias ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      2      2      2      0        0
             asserts      2      2      2      0      n/a
```

Funciones de inicialización y limpieza

Una cosa que no está tan copada es tener que abrir el archivo en cada test, además de que deja un archivo *prueba.txt* que no se borra nunca. Si agregáramos otro test más que use ese *prueba.txt*, no haría lo que nosotros esperamos.

Hagamos un **setup** que borre el archivo y lo cree de vuelta, y una función de limpieza que lo borre (asi al terminar de correr el suite no queda el archivo suelto en la carpeta).

```c
// ...

static char *path = "prueba.txt";
static FILE *archivo = NULL;

int inicializar() {
    unlink(path); // borra el archivo
    archivo = fopen(path,"w+");
    return archivo != NULL ? 0 : -1;
}

int limpiar() {
    fclose(archivo);
    return unlink(path);
}

// ...
```

Estas funciones devuelven 0 en caso correcto y "distinto de 0" en caso contrario.

Agregando un caso más, quedaría algo como:

```c
#include "lector.h"
#include <CUnit/Basic.h>

static char *path = "prueba.txt";
static FILE *archivo = NULL;

int inicializar() {
    unlink(path); // borra el archivo
    archivo = fopen(path,"w+");
    return archivo != NULL ? 0 : -1;
}

int limpiar() {
    fclose(archivo);
    return unlink(path);
}

void test_contar_devuelve_menos1_si_el_archivo_no_existe() {
    int cantidad = archivo_contar("askjd.txt", 'f');
    CU_ASSERT_EQUAL(cantidad,-1);
}

void test_contar_devuelve_el_numero_exacto_de_ocurrencias() {
    inicializar();
    fprintf(archivo, "aca hay como 4 as");
    fflush(archivo);
    int cantidad = archivo_contar(path,'a');
    CU_ASSERT_EQUAL(cantidad,4);
    limpiar();
}

void test_contar_devuelve_0_si_no_hay_ocurrencias() {
    inicializar();
    fprintf(archivo,"aca hay como 4 as");
    fflush(archivo); //para que se guarde en disco ahora
    int cantidad = archivo_contar(path,'a');
    CU_ASSERT_EQUAL(cantidad,4);
    limpiar();
}

int main() {
    CU_initialize_registry();

    CU_pSuite prueba = CU_add_suite("Archivo", NULL, NULL);
    CU_add_test(
        prueba,
        "archivo_contar devuelve -1 si el archivo no existe",
        test_contar_devuelve_menos1_si_el_archivo_no_existe
    );
    CU_add_test(
        prueba,
        "archivo_contar devuelve el numero exacto de concurrencias",
        test_contar_devuelve_el_numero_exacto_de_ocurrencias
    );
    CU_add_test(
```

```
        prueba,
        "archivo_contar devuelve 0 si no hay concurrencias",
        test_contar_devuelve_0_si_no_hay_ocurrencias
    );

    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();

    return CU_get_error();
}
```

(de paso le cambié el nombre al suite de "Suite de prueba" a "Archivo", que refleja más lo que se está probando)

```
Suite: Archivo
  Test: archivo_contar devuelve -1 si el archivo no existe ...passed
  Test: archivo_contar devuelve el numero exacto de concurrencias ...passed
  Test: archivo_contar devuelve 0 si no hay concurrencias ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      3      3      3      0        0
             asserts      3      3      3      0      n/a
```

> Che, pero estás llamando a `inicializar()` y `limpiar()` por cada test. ¿Eso no lo hacía la biblioteca al llamar a `CU_add_suite()` ?

Lo que te deja hacer la librería es definir una función de inicialización y una de finalización *globales*, no un setup para cada test, por lo que se ejecutarían una sola vez al inicio y al final del suite.

La última versión hasta ahora de CUnit (2.1-2) no soporta funciones de setup o teardown, así que no queda otra que hacer eso por cada test case.

Otros tipos de assert

Todos empiezan con `CU_ASSERT` , así que lo más fácil es escribir eso, `Ctrl` + `Space` , y que el IDE te sugiera el resto. Algunos posibles son:

- `CU_ASSERT_TRUE(value)` : Verifica que una expresión sea verdadera
- `CU_ASSERT_FALSE(value)` : Verifica que una expresión sea falsa

- `CU_ASSERT_EQUAL(actual, expected)` : Verifica que actual == expected
- `CU_ASSERT_NOT_EQUAL(actual, expected)` : Verifica lo opuesto al anterior

- `CU_ASSERT_STRING_EQUAL(actual, expected)` : Verifica que dos strings sean equivalentes [3]
- `CU_ASSERT_STRING_NOT_EQUAL(actual, expected)` : Verifica lo opuesto al anterior
- `CU_ASSERT_NSTRING_EQUAL(actual, expected, count)` : Verifica que los primeros `count` caracteres de las cadenas coincidan.

- `CU_ASSERT_PTR_EQUAL(actual, expected)` : Verifica que los punteros sean equivalentes
- `CU_ASSERT_PTR_NOT_EQUAL(actual, expected)` : Verifica lo opuesto al anterior

- `CU_ASSERT_PTR_NULL(value)` : Verifica que un puntero es `NULL`
- `CU_ASSERT_PTR_NOT_NULL(value)` : Verifica lo opuesto al anterior

Es importante aclarar que **ninguno de estos assert termina con la ejecución del test case si fallan**. Para lograr eso, existen otro grupo de aserciones que se llaman igual pero con el sufijo `_FATAL` [4]:

```c
void test1() {
    CU_ASSERT_TRUE(false);
    printf("Esto se ejecuta!!!1");
}

void test2() {
    CU_ASSERT_TRUE_FATAL(false);
    printf("Esto nunca se ejecuta!!!");
}
```

Preguntas frecuentes

> Mi programa a testear tiene una función `main()` , pero el proyecto de CUnit también tiene una función `main()` ... mi cerebro tira SEGMENTATION FAULT.

En el ejemplo de recién, para simplificar, pusimos el código de dominio y los tests en el mismo proyecto. Esto no es muy viable ya que el proyecto de CUnit es también un ejecutable, y no puede haber dos main() en el mismo proyecto. Acá hay dos opciones:

1- Lo más prolijo es tener todo el código de dominio (y que se pretenda testear) en una static library, y que tanto el proyecto como el proyecto de tests usen esa static library.

2- Tener un main que "switchee" los dos, y que con algún parámetro definido se corran los tests de CUnit:

```c
int main(int argc, char** argv) {
    if (strcmp(argv[1], "-corremeLosTests") == 0) {
        return correrTests();
    }

    //código del programa
}

int correrTests() {
    //maindeCUnit
}
```

> Todo muy lindo, pero vos me estás diciendo que haga tests unitarios para probar mi trabajo práctico. El tiempo que toma desarrollar el TP es mucho, apenas llego a la primera instancia de evaluación con horas de diferencia, ¡¿y encima vos me decís que pierda el tiempo haciendo tests?!

En un primer lugar, hacer tests unitarios no es perder el tiempo, estamos probando cada funcionalidad por separado y de manera automatizada.

En un mundo feliz deberías programar los tests para todas las funcionalidades de tu programa, pero sabemos que tiempo no te sobra y entonces te recomendamos que, por lo menos, programes los tests de lo que consideres **importante** en tu programa.

Creeme, sentarte a hacer los tests en un par de horas para después correrlos cada vez que haya cambios importantes en tu código va a minimizar las "sorpresas" que te podés llevar el día de la entrega y vas a solucionar los errores mucho más rápido que haciendo debugging a mano.

---

1. El modo verboso es necesario para que muestre qué test está corriendo en cada momento. Es importante porque si alguno hace que el programa explote (segmentation fault, o similar) los demás tests no se llegan a ejecutar. ↩
2. Para más información, revisar la documentación ↩
3. equivalentes: en caso de strings, internamente realiza `strcmp()` ; en caso de punteros, compara el valor al que apuntan. ↩
4. Para más info, podés revisar la documentación oficial ↩