

Unit Testing con CSpec

Introducción

¿Qué es CSpec?

Es una biblioteca para hacer pruebas unitarias a programas escritos en C, que nos sirven para tener una mayor visión de la robustez del código y asegurar que al menos lo que está testeado anda.

Al definir las pruebas como **test cases**, se pueden repetir de 1 a N veces, validando 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 estás inspirado) 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 tests. 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, ¿cómo lo instalo?

Primero tenés que instalarlo. **Esto tenés que hacerlo una sola vez por máquina.** Si ya lo tenés instalado, no hace falta 😊 Ejecutá en una terminal los siguientes comandos:

```
git clone https://github.com/mumuki/cspec.git
cd cspec
make
sudo make install
```

bash

¿Cómo funciona?

Antes que nada, vamos a definir algunos conceptos:

- **Test case:** Es un bloque de código destinado a probar una funcionalidad. Idealmente se compone de 3 partes:
 - *Inicialización:* generar el fixture de datos a usar durante la prueba.
 - *Operación:* llamar a las funciones que hagan falta.
 - *Aserciones:* validar que los resultados sean los esperados.
- **Suite:** Es un conjunto de test cases que prueban una funcionalidad común. Opcionalmente, además, puede contener funciones de inicialización/limpieza a correr

antes/después de cada test.

- **Context:** Es una agrupación de suites.

Ejemplo en CSpec

```
#include <cspecs/cspec.h>
#include <string.h>

context (test_de_strings) {
    describe("strlen") {
        it("devuelve la longitud de la cadena") {
            char* unaCadena = "Empanada";
            int longitud = strlen(unaCadena);
            should_int(longitud) be equal to(8);
        } end
        it("devuelve 0 para una cadena vacía") {
            should_int(strlen("")) be equal to(0);
        } end
    } end
}
```

Podemos ver que en el **contexto** `test_de_strings`, hay una suite para testear `strlen` que tiene dos test cases. Obviamente uno nunca va a querer testear funciones de la biblioteca estándar de C ¡porque se supone que andan!, pero es solo un ejemplo para que te familiarices con la sintaxis.

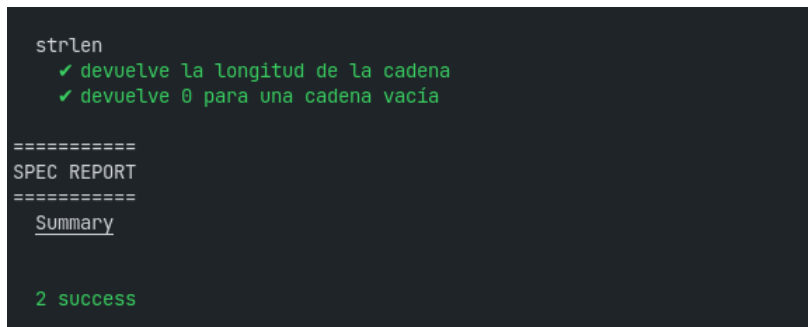
Para correr este ejemplo, copió el código en un `prueba.c` y compilalo con:

```
gcc prueba.c -o prueba -lcspecs
```

Luego, ejecutalo con:

```
./prueba
```

Vas a ver algo como:



```
strlen
  ✓ devuelve la longitud de la cadena
  ✓ devuelve 0 para una cadena vacía

=====
SPEC REPORT
=====
  Summary

2 success
```

O sea, que todas tus pruebas pasaron.

¿Y si hubieras escrito mal el test? Si el segundo test fuera...

```

it("devuelve 0 para una cadena vacía") {
    should_int(strlen("")) be equal to(123);
} end

```

... el resultado sería:

```

strlen
✓ devuelve la longitud de la cadena
1) strlen - devuelve 0 para una cadena vacía

=====
SPEC REPORT
=====
Summary

1) strlen - devuelve 0 para una cadena vacía
   - Expected <123> but was <0> [src/example.c:12]

1 success
1 failure

```

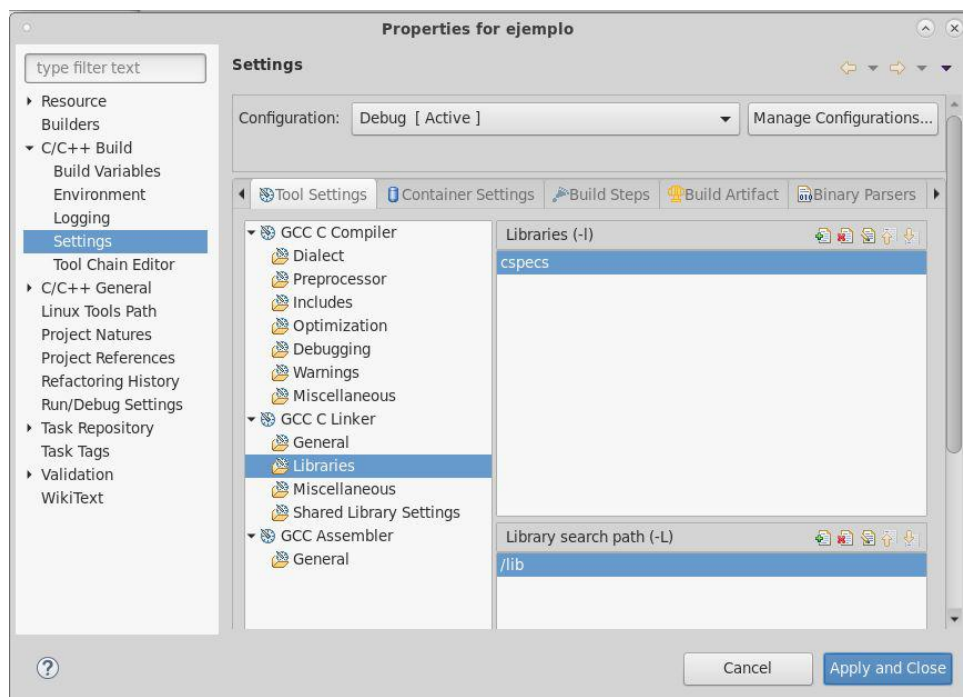
Y claro, el segundo test falló, esperaba que devuelva 123 pero devolvió 0.

Configurando el proyecto de Eclipse

Para dejar de probar con **gcc** y usar nuestros tests en un proyecto de Eclipse, tenemos que *linkear* con la biblioteca **cspecs** ⇒ Hagamos eso:

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

...y agregamos la biblioteca **cspecs** en **Libraries**.



Ejemplo 1: Orden, inicialización y limpieza

Veamos este otro ejemplo, en donde agregamos una función de **inicialización**, otra de **limpieza**, y un test que **rompe**:

```
#include <cspecs/cspec.h>
#include <stdio.h>
#include <string.h>

context (probando_cosas) {
    describe("tests") {
        before {
            printf("Yo inicializo cosas\n");
        } end

        after {
            printf("Yo limpio cosas\n");
        } end

        it("test1") {
            printf("Soy el test 1 y pruebo que 1+1 sea 2\n");
            should_int(1 + 1) be equal to(2);
        } end

        it("test2") {
            printf("Soy el test 2 y doy Segmentation Fault\n");
            char* puntero = NULL;
            *puntero = 9;
        } end

        it("test3") {
            printf("Soy el test 3");
        } end
    } end
}
```

Si lo compilás y ejecutás como ya sabés, vas a ver algo como esto:

```
tests
Yo inicializo cosas
Soy el test 1 y pruebo que 1+1 sea 2
✓ test1
Yo limpio cosas
Yo inicializo cosas
Soy el test 2 y doy Segmentation Fault
[1] 27898 segmentation fault (core dumped) ./bin/tests.out
```

Con esto descubrimos que:

- El bloque **before** se ejecuta antes de cada test dentro de la misma suite.
- El bloque **after** se ejecuta después de cada test dentro de la misma suite.
- Si un test rompe (tira `segmentation fault`) aborta la ejecución de los demás.
- Los tests corren en orden, aunque tu código nunca debería confiar en esto. Los tests tienen que ser independientes entre sí, y cualquier efecto de lado que tengan debe ser

limpiado en el bloque **after** y reinicializado en el **before** para que el siguiente test tenga el mismo contexto de ejecución.

Ejemplo 2: Lector de archivos

Supongamos que hicimos un par de funciones de manejo de archivos y las queremos testear. El código a probar está a continuación. Básicamente es una función que cuenta el número de ocurrencias de un determinado carácter en un archivo.

```
lector.h  lector.c

#ifndef LECTOR_H_
#define LECTOR_H_

int archivo_contar(char* path, char c);

#endif
```

Si tuviéramos un archivo **algo.txt** con el 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. Incluyamos el header a nuestro archivo principal (**main.c** del proyecto, supongamos):

```
#include "lector.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <specs/cspec.h>
```

Para arrancar, podemos probar la primer parte, cuando el archivo que recibe como parámetro no existe. Armamos un test case para eso, lo agregamos a un **context** y vemos que pasa correctamente:

```
#include "lector.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <specs/cspec.h>

context (probando_cosas) {
    it("devuelve -1 si el archivo no existe") {
        int cantidad = archivo_contar("askjd.txt", 'f');
        should_int(cantidad) be equal to(-1);
    } end
}
```

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

```
#include "lector.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <cspecs/cspec.h>

context (probando_cosas) {
    it("devuelve -1 si el archivo no existe") {
        int cantidad = archivo_contar("askjd.txt", 'f');
        should_int(cantidad) be equal to(-1);
    } end

    it("devuelve el número exacto de ocurrencias") {
        FILE* archivo = fopen("prueba.txt", "w+");
        fprintf(archivo, "aca hay como 4 as");
        fflush(archivo); // para que se guarde en disco ya
        int cantidad = archivo_contar("prueba.txt", 'a');
        should_int(cantidad) be equal to(4);
    } end
}
```

¿Lo corremos?

```
✓ devuelve -1 si el archivo no existe
1) - devuelve el número exacto de ocurrencias

=====
SPEC REPORT
=====
Summary

1) - devuelve el número exacto de ocurrencias
    - Expected <4> but was <3> [src/example.c:16]

1 success
1 failure
```

Nos dice que falló, en dónde, y con qué error. A ver qué pasó...

¡Uh!, estábamos recorriendo mal el array, desde 1 en lugar de desde 0:

```

#include "lector.h"

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>

int obtener_size(char* path) {
    struct stat stat_file;
    stat(path, &stat_file);
    return stat_file.st_size;
}

char* leer(char* path) {
    FILE* archivo = fopen(path, "r");
    if (archivo == NULL) {
        return NULL;
    }
    int size = obtener_size(path);

    char* texto = malloc(size + 1);
    fread(texto, size, sizeof(char), archivo);
    fclose(archivo);
    texto[size] = '\0';

    return texto;
}

int archivo_contar(char* path, char c) {
    char* contenido = leer(path);
    if (contenido == NULL) {
        return - 1;
    }

    int cantidad = 0;
    for (int i = 1; i < strlen(contenido); i++) {
        if (contenido[i] == c) {
            cantidad++;
        }
    }

    return cantidad;
}

```

¡Genial! Encontramos un bug gracias al test. Luego de arreglarlo, vemos que está todo bien.

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. Además, si nuestro `archivo_contar(...)` de alguna forma mutara el archivo, los demás tests no harían lo que esperamos. Hagamos un **before** que cree el archivo y un **after** que lo borre:

```

#include "lector.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <cspecs/cspec.h>

context (probando_cosas) {
    describe("con archivos inexistentes") {
        it("devuelve -1 si el archivo no existe") {
            int cantidad = archivo_contar("askjd.txt", 'f');
            should_int(cantidad) be equal to(-1);
        } end
    } end

    describe("con archivos que existen") {

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

        before {
            archivo = fopen(path, "w+");
        } end

        after {
            fclose(archivo);
            unlink(path);
        } end

        it("devuelve el número exacto de ocurrencias") {
            fprintf(archivo, "aca hay como 4 as");
            fflush(archivo); // para que se guarde en disco ya
            int cantidad = archivo_contar("prueba.txt", 'a');
            should_int(cantidad) be equal to(4);
        } end
    } end
}

```

Fijate que que dividimos todo en dos suites según si usaban un archivo o no, quedó mucho más prolijo 😊

Algunos detalles

- En CSpec, los bloques `before` y `after` **siempre** tienen que estar arriba de los test cases, justo abajo de comenzar el bloque `describe`. No ponerlos al principio puede causar problemas extraños.
- ¡Hay muchos tipos de aserciones! En estos ejemplos solo usamos `should_int` pero podés ver el resto en [la documentación de CSpec](#).

- La **so-commons-library** usa CSpec para probar sus funcionalidades. Podés ver más ejemplos de cómo testear en **la carpeta unit-tests del repositorio**.
- A diferencia de frameworks de test de otras tecnologías, acá ninguna de las aserciones termina con la ejecución del test case si fallan.

Preguntas frecuentes

Todo esto no parece código C... ¿Qué es esa sintaxis rara de `} end` ?

~~No importa, vos fumá.~~ Son **macros de preprocesador** definidas en las bibliotecas, creadas para que puedas definir los tests de forma más simple. Son bastante útiles para evitar escribir código repetitivo.

¿Cómo puedo organizar mejor los tests?

Lo conveniente sería tener un `context` por cada archivo a testear, y cada uno de ellos en un archivo diferente.

¿Pueden convivir los tests con mi función `main` ?

Sí, y al ejecutar el programa van a correr las dos cosas (tu main y los tests). Lo ideal sería tener todo el código de dominio (y que se pretenda probar) en una **static library** y que tanto el proyecto principal como el que contengan tus tests usen esa biblioteca.

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.