Gonzalo Almeida - Cátedra Sistemas Operativos 2008  
 Revisado por Lord Hobborg - Cátedra Sistemas Operativos 2010  
 Revivido por Milagro Teruel - Cátedra Sistemas Operativos 2019

# **TL;DR**

(too long; didn't read) Para ayudarlos en el desarrollo, les proveemos una batería de test para distintos módulos del sistema. Para ejecutar los tests, pueden usar los comandos descriptos en el archivo test/Makefile.

### **Preguntas Frecuentes**

1. **Pregunta: Corro make test y ¡no anda! ¿Qué se rompió?** **Respuesta:** Revisa que tengas instalado la biblioteca "check". En sistemas Debian y Ubuntu, hace falta instalar el paquete llamado check. Si no la tenés instalada el mensaje de error al compilar va a incluir algo parecido a: error: check.h: No such file or directory.
2. **Pregunta: Tengo RedHat/SuSE/Fedora/Mandriva, e instale un paquete que se llamaba libcheckAlgunnúmero, pero igual me dice que sigue faltando check.h ¿Qué me falta?** **Respuesta:** Busca un paquete que se llama igual al que instalaste, pero que al final tiene un sufijo -dev o -devel (por ejemplo libcheck0-devel). Instala *también* ese.
3. **P: Cuando corro los tests aparecen un monton de mensajes de assert fallados. ¿Es normal? ¿Hay algo mal?** **R:** Los tests que entregamos, entre varias otras cosas revisan que haya asserts en algunos lugares importantes. Cuando hace el chequeo, assert va a imprimir un mensaje de error, pero Check va a marcar el test como satisfactorio (si NO salta alguno de los assert que correspondería, ese test se va a considerar fallido). Pueden ignorar tranquilos los mensajes de assert que salen mientras los tests corren. Lo que importa es la línea de resultado 96%: Checks: 131, Failures: 5, Errors: 0 y el resumen que sigue después. Si en ese línea dice "100%" pueden quedarse tranquilos de que los tests pasan, sin importar los mensajes de antes. Mas detalles en un [mensaje de años anteriores](http://www.famaf.proed.unc.edu.ar/mod/forum/discuss.php?d=216#p698) y en la [documentación de check.](http://check.sourceforge.net/doc/check.html/check_15.html#SEC17)
4. **P: Me fallan un montón de tests con el mensaje Test timeout expired, expected signal 6 (Aborted). ¿Qué significa?** **R:** Algunos programas (como el Midnight Commander, mc), corren otros programas dentro de una forma especial que impide que Check funcione. Probá correr los tests desde una terminal donde solo tengas el shell y nada más cargado, y se debería resolver.
5. **P: Terminé el proyecto, pero no me pasan todos los tests, al 100%. ¿Importa?** **R:** Sí. Eso quiere decir que tu programa parece que anda, pero en algunos de los casos que los tests cubren, no anda del todo. Fijate cuales son los nombres de los tests que fallan al final de la salida de make test. Después, busca en el directorio tests/ la funcion con ese nombre. En esa función vas a ver que funciones esta llamando el test, y que resultado esperaba obtener.
6. **P: Creo que tengo algún otro problema. ¿Qué hago?** **R:** Preguntá en la lista de docentes. Si hay algo que te da un mensaje de error, copia *todo* el mensaje de error al preguntar.

A continuación, les dejamos una guía de qué es testing y por qué es una MUY buena práctica de la programación. Además, un poco de documentación sobre cómo están escritos los test que les damos.

# **Introducción a Unit Testing**

Los programadores somos seres humanos y, por tal razón, cometemos errores. El desarrollo de software no es la excepción, especialmente miestras estamos aprendiendo. Todo programa de computadora tiene desperfectos, es decir, un comportamiento distinto al esperado, que es causado por un defecto o bug. Puede verse así: el defecto sería una cadena de transmisión de mala calidad, el desperfecto sería cuando la cadena se corta y el auto ya no se quiere mover.

Si ocurre un desperfecto significa que había un defecto. O sea, si hay un comportamiento distinto al especificado, hay un bug en el programa. Garantizar la corrección de un programa como un todo es muy difícil debido a la excesiva complejidad del mismo, desde la concepción de la idea hasta su implementación, lo que multiplica las oportunidades de error.

## **¿Qué es testing?**

Se denomina **Testing** al hecho de poner a prueba un programa, y por extensión se llama también así a los procesos que permiten verificar y revelar la calidad de un producto de software.  
 Un programa suele ser algo complejo y para ser considerado de "buena calidad" debe cumplir con muchos requisitos. Uno de los principales es que presente pocos bugs. Por ello antes de hacer un lanzamiento oficial (o una entrega de un Lab) todo programa se somete a prueba mediante Testing a modo de ser evaluado por la cantidad de errores encontrados.

Durante el testing un programa se ejecuta siguiendo un conjunto de casos de test. Si hay desperfectos en la ejecución de un test significa que hay defectos en el software. Y como se dijo en clase si no ocurren desperfectos la confianza en el software crece, pero difícilmente podamos asegurar la ausencia total y absoluta de bugs.

| “Program testing can be used to show the presence of bugs, but never to show their absence!”  Edsger Wybe Dijkstra |
| --- |

Por eso queremos que cualquier defecto que tenga nestro código sea bien evidente, para que lo veamos y podamos arreglarlo antes de hacer la entrega definitiva. Y para poner de manifiesto los bugs usamos el “unit testing”.

## **Pruebas de unidad (unit testing)**

Es una forma de probar el funcionamiento de un módulo del código. Lo que haremos es asegurar que cada uno de los módulos funcione correctamente por separado. Luego con los "tests de integración" se podrá asegurar el correcto funcionamiento de todo el sistema o subsistema en cuestión.  
 El enfoque del unit testing concibe al software como un conjunto de unidades mínimas fáciles de verificar. Cada test de unidad aisla una parte del programa y le hace pruebas para verificar su corrección.

### **Casos de test**

Se denomina “caso de test” a un conjunto finito de "entradas" con las que se alimenta un programa o módulo para verificar si cumple con lo especificado.  
 La idea es ejecutar el módulo con esas entradas conocidas y evaluar el valor devuelto. Si es el que esperábamos el módulo pasó el caso de test. Sino habrá que ver qué pasó y porqué.  
 Es importante hacer pruebas con entradas que consideremos cotidianas y también con "casos extremos". Un ejemplo típico de lo segundo es pasarle un NULL a un módulo que toma un string.

### **Test suites**

Se denomina "test suite" a un conjunto de casos de test pensados para evaluar si un programa tiene cierto comportamiento específico. Así como el caso de test se asociaba con un único módulo, un suite se asocia con una funcionalidad determinada del programa.  
 Los test suites son usados para agrupar casos de test similares, junto con su documentación y las metas de cada uno de ellos.

### **¿Por qué escribir test?**

* **Fomentan el cambio:** los test de unidad facilitan que el programador cambie el código para mejorar su estructura, puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores (regression testing)
* **Simplifica la integración:** Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código esta funcionando correctamente. De esta manera se facilitan las pruebas de integración.
* **Documenta el código:** los propios tests sirven como documentación del código puesto que allí se puede ver como utilizarlo.
* **Separación de la interfaz y la implementación:**dado que la única interacción entre los casos de test y las unidades bajo prueba son las interfaces de los módulos, se puede cambiar cualquiera de los dos sin afectar al otro.

### **Limitaciones**

Es importante tener en cuenta que los test de unidad no descubrirán todos los errores del código. Por definición, solo prueban las unidades aisladamente. Por lo tanto no descubrirán errores de integración, problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto. Además puede no ser trivial anticipar todos los casos críticos que en realidad la unidad del programa bajo estudio puede recibir como entrada.

Ahora sabemos que debemos escribir test, pero al parecer es una tarea tediosa que lleva tiempo.  
 Muchos programadores no realizan sus test y la respuesta clásica es: “no tengo tiempo”, “o estoy apurado”. Esto se vuelve rápidamente en un circulo vicioso, por que mientras más presión tenemos en terminar, menos test escribimos, mientras menos test escribamos, seremos menos productivos y nuestro código será menos estable, y por esta razón sentimos más presión aún.  
  
 La mejor manera de convencerse del valor de escribir sus propios test es sentarse y escribir un poco de código. Encontrar nuevos errores por medio de los tests y arreglarlos. Luego si han regresado, arreglarlos de nuevo, y así sucesivamente. Se verá el valor de la información inmediata que se obtiene de la escritura y corrida de su propia unidad de tests.

### **Unit testing framework**

Son bibliotecas o programas para verificar otros programas, que ayudan a simplificar el “proceso de unit testing".  
  
 **¿Por qué usar un framework?**

* **El framework está diseñado para realizar tests:** Si bien es posible realizar unit testing sin la ayuda de un framework específico, esto limitaría mucho el alcance de nuestros tests. Si escribimos a mano un programita que ejercite nuestras unidades usando aserciones y excepciones hay características avanzadas de testeo (que un framework adecuado nos brinda) que faltarían o deberían ser escritas a mano.
* **Lo único que hace falta es ejecutar un comando:** una vez escrito los test, solo debemos dejarlos correr, y al terminar verificar la salida. No hace falta un humano en el medio. Por esta razón no hay excusa para no correrlos.
* **Ayuda a correr los test más frecuentemente:** teniendo los test escritos y automatizados, no hay pérdida de tiempo en correrlos.

Escribir tests nos da mayor confianza, por que no le tendremos miedo al cambio o a romper cosas viejas o miedo a que las cosas nuevas no funcionen, así que podremos jugar y experimentar sin preocuparnos.

## **Check**

Check es un Framework para unit testing en C. este Framework nos facilita la tarea de hacer test.

### **¿Cómo escribir tests en C con check?**

Lo primero que debemos hacer es incluir la biblioteca check.h.

#include <check.h>

...

### **Caso de test:**

Un caso de test (test básico de unidad) se ve más o menos así:

START\_TEST (test\_name)

{

/\* código del test \*/

}

END\_TEST

Los macros START\_TEST/END\_TEST configuran las estructuras básicas que permiten el testing. Es un grave error olvidarse el END\_TEST, producirá muchos errores extraños cuando el test es compilado.

Supongamos que queremos crear nuestro módulo dinero. Por lo tanto tendríamos un archivo dinero.h, en el cual especificaremos la interfase del módulo.

#ifndef DINERO\_H

#define DINERO\_H

typedef struct Dinero Dinero;

Dinero \*dinero\_create (int cantidad, char \*moneda);

int dinero\_cantidad (Dinero \* d);

char \*dinero\_moneda (Dinero \* d);

void dinero\_free (Dinero \* d);

#endif /\* MONEY\_H \*/

Nuestro dinero.c sería de esta forma por ahora:

#include “dinero.h”

Dinero \*dinero\_create(int cantidad, char \*moneda)

{

return NULL;

}

int dinero\_cantidad(Dinero \* d)

{

return 0;

}

char \*dinero\_moneda(Dinero \* d)

{

return NULL;

}

void dinero\_free(Dinero \* d)

{

return;

}

int main(void)

{

return 0;

}

Ahora dentro de una carpeta tests/ incluiremos un archivo test\_dinero.c

#include <check.h>

#include "../dinero.h”

START\_TEST (test\_dinero\_create)

{

Dinero \*d = NULL;

d = dinero\_create (5, "USD");

fail\_unless (dinero\_cantidad (m) == 5,

"La cantidad no fue correctamente establecida en la creación");

fail\_unless (strcmp (dinero\_moneda (m), "USD") == 0,

"La moneda no fue correctamente establecida en la creación");

money\_free (m);

}

END\_TEST

Aquí vemos un caso de test. Check nos ofrece varias funciones (en realidad son macros), para verificar la salida de las interfaces de nuestro módulo. Algunos de ellos son:

* fail\_unless: toma un argumento booleano y algún texto para mostrar en el caso que el argumento boolenao sea falso.
* fail\_if: parecido a fail\_unless solo que falla en el caso que el argumento booleano sea verdadero.

### **Creando un test suite**

Para correr los test con Check primero debemos crear nuestros casos de test y luego le damos forma a nuestro test suite que los agrupa. Para ello agregamos al final de nuestro test\_dinero.c las siguientes líneas:

Suite \*dinero\_suite(void)

{

Suite \*s = suite\_create ("Dinero");

/\* Agregamos los casos de test \*/

TCase \*tc\_core = tcase\_create ("Core");

tcase\_add\_test (tc\_core, test\_dinero\_create);

suite\_add\_tcase (s, tc\_core);

return s;

}

El archivo test\_dinero.c es fácil de entender, estamos creando los casos de test (en nuestro ejemplo es uno solo) y luego los agrupamos dentro de un suite  
 Ahora creamos test\_dinero.h

ifndef TEST\_DINERO\_H

#define TEST\_DINERO\_H

#include <check.h>

#include "dinero.h"

Suite \*dinero\_suite (void);

#endif

Tan simple como eso. Una vez escritos los casos de test y creado el test suite, necesitamos un main que corra los test:

#include <check.h>

#include "test\_dinero.h"

int main (void)

{

int num = 0;

Suite \*s = dinero\_suite ();

SRunner \*sr = srunner\_create (s);

srunner\_run\_all (sr, CK\_NORMAL);

num = srunner\_ntests\_failed (sr);

srunner\_free (sr);

return (num == 0) ? 0 : 1;

}

Aquí estamos agregando el suite que creamos en test\_dinero, para que sea ejecutado cuando corramos los test.  
 Ahora con un simple Makefile podremos compilar nuestro módulo y correr los test.  
  
 Y resulta que... ¡los tests fallan! Pero tranquilos, esa era la idea porque aún no hemos implementado las interfaces de la unidad. Entonces laburamos un poquito dejando a nuestro dinero.c de la siguiente forma:

#include <stdlib.h>

#include "dinero.h"

struct Dinero {

int cantidad;

char \*moneda;

};

Dinero \*dinero\_create (int cantidad, char \*moneda)

{

Dinero \*d = malloc (sizeof (Dinero));

if (d == NULL)

return NULL;

d->cantidad = cantidad;

d->moneda = moneda;

return d;

}

int dinero\_cantidad (Dinero \* d)

{

return d->cantidad;

}

char \*dinero\_moneda (Dinero \* d)

{

return d->moneda;

}

void dinero\_free (Dinero \* d)

{

free (d);

d = NULL;

return;

}

Ahora que tenemos nuestra implementación de el módulo dinero podemos volver a correr los test:

flecox@dingo:~/facu/sistop/testing$ make test

./runner

Running suite(s): Dinero

100%: Checks: 1, Failures: 0, Errors: 0

make[1]: se sale del directorio `/home/flecox/facu/sistop/testing/test'

flecox@dingo:~/facu/sistop/testing$

En la salida podemos ver que nuestro módulo a pasado el único caso de test que escribimos.

### **Testing clásico y Test Driven Development**

Hay dos ideas para hacer unit testing:  
 La primera es: escribir un poco de código y luego testearlo. Éste es el método clásico.  
 La segunda es: primero escribir el test para que falle, y luego escribir el código para que pase el test. Esta metodología es conocida como [TDD](http://www.dosideas.com/wiki/Test_Driven_Development), test driven development (desarrollo dirigido por test). La idea fundamental de TDD esta en especificar el problema por medio de casos de test, y luego implementar la solución para que pase el test.