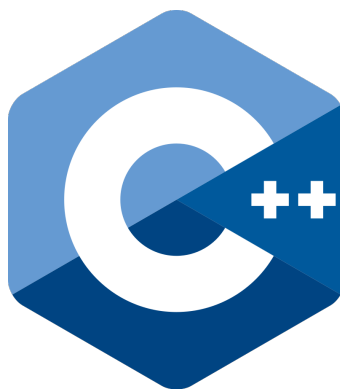




# Metodología de la Programación

DGIM

Curso 2021/2022



## Guion de prácticas

*Test Driven Development*

*Google Test & MPTTest*

*Febrero de 2022*



# Índice

<b>1. Introducción</b>	<b>7</b>
1.1. Test Driven Development . . . . .	7
<b>2. TDD con GoogleTest</b>	<b>9</b>
2.1. Principales herramientas de GoogleTest: definición de tests unitarios . . . . .	9
<b>3. Integración con NetBeans</b>	<b>13</b>
3.1. Preparandolo todo antes de empezar . . . . .	13
3.1.1. El espacio de trabajo . . . . .	13
3.1.2. Configurar el proyecto en NetBeans . . . . .	14
3.1.3. Configurar el entorno de testeo . . . . .	16
<b>4. Extensión de GoogleTest para la definición de tests de inte- gración: MPTests</b>	<b>17</b>
4.1. Saltar un test . . . . .	17
4.2. Tests de integración . . . . .	19
4.2.1. Definir el entorno de ejecución controlado para testear la aplicación completa . . . . .	19
4.2.2. Definiendo checkpoints en la salida . . . . .	20
4.2.3. Aplicaciones embucladas . . . . .	22
4.2.4. Validando el contenido de ciertas clases . . . . .	22
4.2.5. Detectando problemas de memoria dinámica . . . . .	22
4.2.6. Paso de parámetros a main . . . . .	25
4.3. Reportes de tests . . . . .	26
4.3.1. Reporte inicial de los tests que hay preparados . . . . .	26
4.3.2. Reporte del resultado de pasar los tests . . . . .	26
<b>5. Propiedades de TDD</b>	<b>27</b>
<b>6. (git) About GoogleTest</b>	<b>29</b>
6.0.1. Announcements . . . . .	29
6.1. Welcome to <b>GoogleTest</b> , Google's C++ test framework! . . .	29
6.1.1. Getting Started . . . . .	29
6.2. Features . . . . .	29
6.3. Supported Platforms . . . . .	29
6.3.1. Operating Systems . . . . .	30
6.3.2. Compilers . . . . .	30
6.3.3. Build Systems . . . . .	30
6.4. Who Is Using GoogleTest? . . . . .	30
6.5. Related Open Source Projects . . . . .	30
6.6. Contributing Changes . . . . .	31
<b>7. (git) GoogleTest Primer</b>	<b>32</b>
7.1. Introduction: Why googletest? . . . . .	32
7.2. Beware of the nomenclature . . . . .	32
7.3. Basic Concepts . . . . .	33
7.4. Assertions . . . . .	33



7.4.1.	Basic Assertions . . . . .	34
7.4.2.	Binary Comparison . . . . .	34
7.4.3.	String Comparison . . . . .	36
7.5.	Simple Tests . . . . .	36
7.6.	Test Fixtures: Using the Same Data Configuration for Multiple Tests . . . . .	37
7.7.	Invoking the Tests . . . . .	39
7.8.	Writing the main() Function . . . . .	40
7.9.	Known Limitations . . . . .	42
<b>8.</b>	<b>Advanced googletest Topics</b>	<b>43</b>
8.1.	Introduction . . . . .	43
8.2.	More Assertions . . . . .	43
8.2.1.	Explicit Success and Failure . . . . .	43
8.2.2.	Exception Assertions . . . . .	43
8.2.3.	Predicate Assertions for Better Error Messages . . . . .	44
8.2.4.	Floating-Point Comparison . . . . .	48
8.2.5.	Asserting Using gMock Matchers . . . . .	49
8.2.6.	More String Assertions . . . . .	50
8.2.7.	Windows HRESULT assertions . . . . .	50
8.2.8.	Type Assertions . . . . .	50
8.2.9.	Assertion Placement . . . . .	51
8.3.	Teaching googletest How to Print Your Values . . . . .	52
8.4.	Death Tests . . . . .	53
8.4.1.	How to Write a Death Test . . . . .	53
8.4.2.	Death Test Naming . . . . .	55
8.4.3.	Regular Expression Syntax . . . . .	56
8.4.4.	How It Works . . . . .	57
8.4.5.	Death Tests And Threads . . . . .	57
8.4.6.	Death Test Styles . . . . .	58
8.4.7.	Caveats . . . . .	58
8.5.	Using Assertions in Sub-routines . . . . .	59
8.5.1.	Adding Traces to Assertions . . . . .	59
8.5.2.	Propagating Fatal Failures . . . . .	60
8.6.	Logging Additional Information . . . . .	63
8.7.	Sharing Resources Between Tests in the Same Test Suite . . . . .	63
8.8.	Global Set-Up and Tear-Down . . . . .	65
8.9.	Value-Parameterized Tests . . . . .	66
8.9.1.	How to Write Value-Parameterized Tests . . . . .	66
8.9.2.	Creating Value-Parameterized Abstract Tests . . . . .	68
8.9.3.	Specifying Names for Value-Parameterized Test Para- meters . . . . .	69
8.10.	Typed Tests . . . . .	70
8.11.	Type-Parameterized Tests . . . . .	71
8.12.	Testing Private Code . . . . .	72
8.13.	“Catching” Failures . . . . .	74
8.14.	Registering tests programmatically . . . . .	75
8.15.	Getting the Current Test’s Name . . . . .	76
8.16.	Extending googletest by Handling Test Events . . . . .	77



8.16.1. Defining Event Listeners . . . . .	77
8.16.2. Using Event Listeners . . . . .	78
8.16.3. Generating Failures in Listeners . . . . .	78
8.17. Running Test Programs: Advanced Options . . . . .	79
8.17.1. Selecting Tests . . . . .	79
8.17.2. Repeating the Tests . . . . .	81
8.17.3. Shuffling the Tests . . . . .	81
8.17.4. Controlling Test Output . . . . .	82
8.17.5. Controlling How Failures Are Reported . . . . .	87
8.17.6. Sanitizer Integration . . . . .	88
<b>9. Código de ejemplo. class MyVector</b>	<b>89</b>
9.1. myvector.h . . . . .	89
9.2. myvector.cpp . . . . .	89
9.3. main.cpp . . . . .	89
<b>10.Changelog</b>	<b>90</b>
<b>11.Videotutoriales</b>	<b>93</b>



## 1. Introducción

El El objetivo de este documento es hacer una breve introducción a la metodología TDD (*Test-Driven Development*) y a su puesta en marcha usando una suite de testeo conocida (Google Test) a la que se le han añadido algunas funciones específicas de esta asignatura, con el doble objetivo de introducir esta metodología y que la experiencia de aprendizaje sea lo más estándar posible para los alumnos.

Hay disponible una serie de videotutoriales sobre este tema que aparecen en la última sección de este documento.

### 1.1. Test Driven Development


TDD es una metodología de desarrollo de aplicaciones que se basa en escribir, en primer lugar, los tests que debe de pasar correctamente el software que se va a crear y, posteriormente, crear ese software con el objetivo fundamental de que pase todos los tests para, posteriormente, adaptarlo a un diseño y/o arquitectura específica. Eso sí, sin dejar de pasar nunca todos los tests. Quizás los dos objetivos fundamentales de TDD sean buscar el diseño más sencillo posible, evitando toda complejidad innecesaria (KISS principle<sup>1</sup>) y, por otro lado, proporcionar confianza en el desarrollo de software.

Hay varios tipos de tests, que varían según la aplicación que se esté desarrollando, entre los que podemos distinguir los más básicos:

- Tests Unitarios<sup>2</sup>. Cuyo objetivo es comprobar que cada unidad funcional mínima que se construya, ya sea una función o un método, cumple una serie de buenas propiedades. Así, podemos considerar que la implementación de una clase es correcta si todos sus métodos han sido probados individualmente, lo cual ofrece una base de trabajo garantizada desde la que desarrollar nuevas clases o construir el programa completo.
- Tests de Integración<sup>3</sup>. Cuyo objetivo es comprobar que la integración de todas las clases construidas, la entrada y salida de datos, sigue siendo correcta, partiendo de la base de que las clases se han probado correctas con tests unitarios.

---

<sup>1</sup>KISS = Keep It Simple Stupid,  (Abrir →)

<sup>2</sup>Unit Testing  (Abrir →)

<sup>3</sup>Integration Testing  (Abrir →)

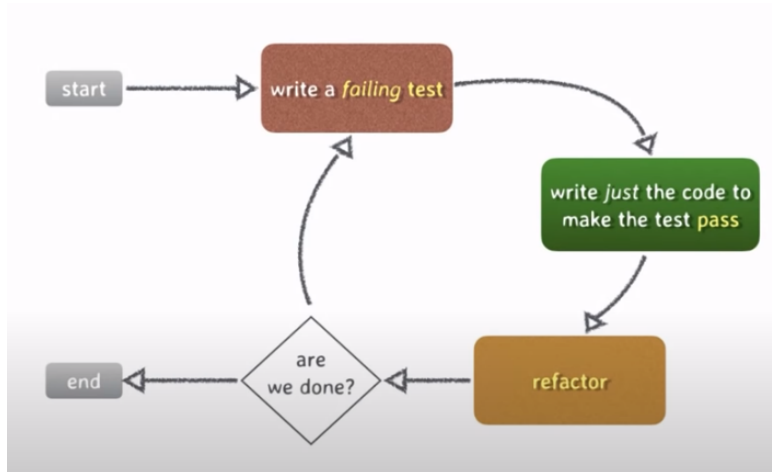


Figura 1: Ciclo de vida de TDD. Phil Nash, CppCon The C++ Conference, 2020.  (Abrir →)

Los pasos a seguir en la metodología TDD, que no son muy diferentes del proceso tradicional, sólo que están más ordenados (ver Figura 1), son los siguientes (en **negrilla** los que debe de hacer el alumno, en tipografía normal los pasos que aporta el profesor).

1. Escribir el diseño de la API de la clase, sólo los métodos.
2. Escribir los tests unitarios y de integración.
3. **Escribir el código de los métodos hasta que pasen todos los tests unitarios, sea como sea.**
4. **Cuando pasen todos los tests, intentar refactorizar el código para mejorarlo, en cuyo caso habría que volver al paso 3 para testear los cambios hechos.**
5. Escribir el `main()` para que pase los tests de integración, sea como sea.
6. Es posible que haya que refactorizar alguna clase previa. Si es así, volver al paso 3.
7. Si ha pasado todos los tests de integración, intentar refactorizar el código para mejorarlo y volver al paso 5.
8. Compilar la versión definitiva.



```
class Number {  
private:  
    int data;  
public:  
  
    Number() { data = 0; }  
    int get() const { return data; }  
    void increase() { data++; }  
    void add(Number n) { data += n.get(); }  
    bool isEven() { return data % 2 == 0; }  
};
```

Figura 2: Fichero `Number.h`

## 2. TDD con GoogleTest

Esta sección describe cómo se puede aplicar esta metodología, en el caso concreto de la asignatura Metodología de la Programación. Para aplicar TDD a un proyecto de desarrollo de software necesitamos dos cosas:

1. En primer lugar, un entorno de testeo, de los que hay muchas opciones, y suelen consistir en una librería de funciones preparadas para realizar los tests sobre nuestro proyecto. En este caso se va a usar la suite de testeo **GoogleTest** desarrollada por Google para testear aplicaciones escritas en C++<sup>4</sup> y se ha ampliado para cubrir también unos sencillos tests que no cubre GoogleTest (extensión **MPTest**).
2. Un entorno de desarrollo en el que integrar el entorno de testeo con nuestro proyecto. De nuevo hay muchas opciones, pero se ha elegido **Apache NetBeans 12**<sup>5</sup>.

### 2.1. Principales herramientas de GoogleTest: definición de tests unitarios


GoogleTest es una suite de testeo muy potente y con muchísimas funcionalidades que se pueden consultar en si Git o en las secciones 6 y sucesivas, extraídas, precisamente, del Git de Googletest, las más básicas de las cuales se describen aquí.

Supóngase la siguiente clase (Figura 2, cuya implementación se ha incluido para hacerla autoexplicativa y más fácil de introducir los tests unitarios).

La programación de tests unitarios pretende validar el funcionamiento de cada método o función sin necesidad de haber programado aún la función `main()` y suelen programarse mediante macros que emulan llamadas a funciones. En este caso se usa la macro `TEST(C,T)` con dos parámetros, el primero es un identificador que representa al conjunto de tests `C` y el segundo, al test unitario en particular `T` dentro de ese conjunto de tests. Normalmente cada clase define un conjunto de tests, y cada método al menos un test individual. Estos tests (Figura 3) se escriben en ficheros `.cpp` que forman parte del proyecto, pero no se almacenan en la carpeta `src` sino en la carpeta `tests`.

---

<sup>4</sup>GoogleTest  (Abrir →)

<sup>5</sup>Apache NetBeans 12  (Abrir →)

```
#include <gtest/gtest.h>
#include "Number.h"

using namespace std;

TEST(C, T) {
    //cuerpo del test
}
```

Figura 3: Fichero de tests, que podría llamarse `Unit_Number.cpp` y que estaría ubicado en la carpeta `tests` y forma parte del proyecto

El cuerpo del test se puede programar como una función más en C++.

```
TEST(Number, Number_increase) {
    Number a;           // Se inicializa a 0
    a.increase();       // Se incrementa en 1
}
```

Con el principal añadido que se realizan aserciones para comprobar propiedades que el método debe cumplir y estas aserciones son macros que se definen en GoogleTest, como esta, a continuación que comprueba la igualdad de dos valores con `ASSERT_EQ`

```
TEST(Number, Number_increase) {
    Number a;           // Se inicializa a 0
    a.increase();       // Se incrementa en 1

    ASSERT_EQ(a.get(), 1);
}
```

Si ambos valores coinciden, se da el test por superado (aparece en verde) y, si fuese el único test, se daría por pasada la fase de test completa. En la Figura siguiente aparece el inicio y el final de toda la fase de test con la marca

[=====]            Inicio y final la fase de testeo

Y cada conjunto o agrupación de tests, lo que se conoce como "test suite", aparece marcado al inicio y al final con la marca

[-----]            Inicio y final de un grupo de tests (suite)

```
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from Number
[ RUN      ] Number.Number_increase
[ OK       ] Number.Number_increase (0 ms)
[-----] 1 test from Number (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[ PASSED  ] 1 test.
```

en otro caso, el test falla (aparece en rojo) y GoogleTest nos explica por qué: “Ha salido 3, cuando esperaba que hubiese salido 1”. En este caso, el fallo de un único test, hace que toda la fase de tests falle también.



```
[.....] Running 1 test from 1 test suite.
[.....] Global test environment set-up.
[.....] 1 test from Number
[ RUN      ] Number.Number_increase
src/main.cpp:48: Failure
Expected equality of these values:
  a.get()
    which is: 3
1
[ FAILED   ] Number.Number_increase (0 ms)
[.....] 1 test from Number (0 ms total)

[.....] Global test environment tear-down
[.....] 1 test from 1 test suite ran. (0 ms total)
[ PASSED   ] 0 tests.
[ FAILED   ] 1 test, listed below:
[ FAILED   ] Number.Number_increase

1 FAILED TEST

RUN FINISHED; exit value 1; real time: 0ms; user: 0ms; system: 0ms
```

En estos casos, se puede añadir un texto explicativo, como si fuese `cout` junto a la aserción.

```
TEST(Number, Number_increase) {
    Number a;          // Se inicializa a 0
    a.increase();       // Se incrementa en 1
    ASSERT_EQ(a.get(), 1) << "Si incrementamos una instancia recién creada de Number, el resultado debe ser 1" << endl;
}
```

```
[.....] Global test environment set-up.
[.....] 1 test from Number
[ RUN      ] Number.Number_increase
src/main.cpp:48: Failure
Expected equality of these values:
  a.get()
    which is: 3
1
Si incrementamos una instancia recién creada de Number, el resultado debe ser 1
[ FAILED   ] Number.Number_increase (0 ms)
[.....] 1 test from Number (0 ms total)
```

El incumplimiento de esta aserción daría por no pasada la fase de test y obligaría a escribir o refactorizar el código para que lo pudiese pasar.

Además de `ASSERT_EQ` existen muchas más aserciones, dependiendo de la propiedad que se quiera comprobar, aparecen todas descritas en las Secciones 6 y 7 y aparecen resumidas en el Cuadro 1.

<code>ASSERT_EQ(expr1, expr2)</code>	Las dos expresiones deben ser iguales
<code>ASSERT_NE(expr1, expr2)</code>	Las dos expresiones deben ser diferentes
<code>ASSERT_TRUE(expr1)</code>	La expresión debe ser verdadera
<code>ASSERT_FALSE(expr1)</code>	La expresión debe ser falsa
<code>ASSERT_STREQ(char*1, char*2)</code>	Las dos cadenas deben ser iguales
<code>ASSERT_STNE(char*1, char*2)</code>	Las dos cadenas deben ser iguales
<code>ASSERT_DEATH(expr1, message)</code>	Al evaluar la expresión, debe saltar un <code>assert</code>
<code>ASSERT_EQ(expr1, expr2)</code>	Las dos expresiones deben ser iguales

Cuadro 1: Algunas de las aserciones más frecuentes de GoogleTest

Aunque todo esto se puede hacer sin definir un `main()`, sí que es necesario inicializar la librería de tests y ordenar la ejecución de los tests, por lo que el `main` realmente quedaría así:



```
#include <gtest/gtest.h>
#include "Number.h"

using namespace std;

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);

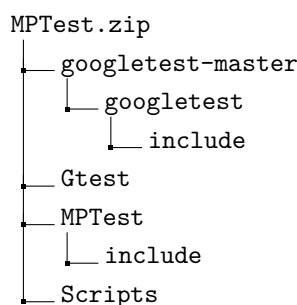
    return RUN_ALL_TESTS();
}
```

### 3. Integración con NetBeans

Esta sección hace un boceto de cómo sería la implementación de TDD en NetBeans, usando para ello GoogleTest, pero se pueden encontrar videotutoriales completos sobre este tema en Google Drive <sup>6</sup>.

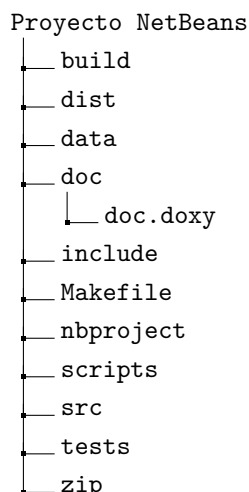
#### 3.1. Preparandolo todo antes de empezar

##### 3.1.1. El espacio de trabajo



Este es el material inicialmente entregado a los alumnos en Prado, clasificado por carpetas. **NetBeans** Carpeta raíz desde la que se van a crear todos los proyectos de la asignatura. **googletest-master** Rama principal de desarrollo de GoogleTest en Git. No es necesario que sea la última. **GTest** Proyecto de NetBeans que permite generar, a cada alumno, una versión enlazable de GoogleTest para su máquina y sistema operativo. **MPTest** Extensión para cubrir los tests de integración en MP. Por ahora esta extensión es un único fichero de cabeceras el cual

tiene las macros necesarias para ello. **Scripts** Una serie de scripts Bash de apoyo a las funciones de NetBeans. Se recomienda empezar por crear la carpeta NetBeans y descomprimir el fichero de material previo **MPTest.zip** dentro de ella, como muestra la figura a la izquierda. Abrir el proyecto **Gtest** y compilarlo (Clean & Build).



Para tener bien ordenado el contenido de cada proyecto NetBeans, se va a dividir en las siguientes carpetas.

**build** Es una carpeta temporal que contiene código precompilado y pendiente de enlazar

**dist** Contiene los binarios ejecutables. Vamos a generar dos versiones por cada programa.

**Release** Es la versión más eficiente y que menos espacio ocupa. Es la que debería entregarse al cliente.

**Debug** Es la versión sobre la que se depuran errores y se corrigen defectos. Es más grande porque el binario contiene información extra para poder usar el depurador.

Figura 4: Estructura interna de cada proyecto NetBeans

<sup>6</sup>Google Drive, Carpeta Test Driven Development  <http://> (Abrir →)



Puede llegar a ser el doble de grande que la anterior.

**data** Ficheros de datos, bases de datos, ficheros de configuración que pueda necesitar el programa durante su ejecución, que no sean para testearla.

**doc** Aquí se almacena la documentación del proyecto, tanto del código, como la generada por los tests.

**include** Contiene los ficheros de cabeceras **.h**.

**nbproject** Es la carpeta que utiliza NetBeans para almacenar la configuración del proyecto

**scripts** En esta carpeta colocaremos las scripts de apoyo a Netbeans que se han desarrollado en esta asignatura y se puede ampliar con otras más.

**src** Contiene los principales ficheros fuente del proyecto **.cpp**, sin incluir los ficheros de test, que también son **.cpp**, que van en la siguiente carpeta

**tests** Carpeta dedicada a los tests unitarios y de integración. Dentro de ella se encuentran, en el primer nivel, los ficheros fuente de los tests **.cpp**, la carpeta **output** que se utiliza como carpeta temporal durante los tests y la carpeta **validation** que se utiliza para **ficheros de datos únicamente usados en la validación de la aplicación** completamente integrada.

**zip** Carpeta para copias de seguridad del proyecto.

### 3.1.2. Configurar el proyecto en NetBeans

Se recomienda repasar primero los videotutoriales completos sobre este tema

en Google Drive  ([Abrir →](#)) y descargar el fichero de material **MyVectorProject.zip** ([Abrir →](#)) cuyo contenido es el siguiente:

```
MyVectorProject.zip
├── 01_Unit_MyVector.cpp (→ ./tests/)
├── 02_Unit_MyVector.cpp (→ ./tests/)
├── 10_Integration_MyVector.cpp (→ ./tests/)
├── documentation.doxy (→ ./doc/)
├── main.cpp (→ ./src/)
├── myvector.cpp (→ ./src/)
├── myvector.h (→ ./include/)
├── ReadmeTests.md (→ ./doc/)
└── scripts (→ ./script/)
```



1. Crear el proyecto en NetBeans como **C++ Application**.
  - a) Eliminar el entorno de trabajo **Release** y dejar sólo **Debug**. Aunque en adelante se crearán otros entornos de trabajo, en lo siguiente se trabajará en **Debug**.
2. Crear la carpeta `./scripts`, copiar la script `runUpdate.sh` dentro y ejecutarla. Esto crea la estructura de carpetas descrita anteriormente y actualiza las scripts que pudiese haber nuevas.
3. Colocar cada fichero en su sitio, según indica la imagen anterior.
4. Propiedades de proyecto. Compilador
  - a) Poner el estándar a C++14
  - b) Añadir los **Include Directories** ... siguientes:  
del propio proyecto  
`./include`  
y las de testeo  
`../MPTest/include`  
`../googletest-master/googletest/include`  
  
y el resto de librerías externas que pueda utilizar el proyecto.
5. Desde la vista lógica del proyecto.
  - a) En **Header Files**, **Add existing item** añadir el fichero `myvector.h`.
  - b) En **Source Files**, **Add existing item** añadir los ficheros `main.cpp` y `myvector.cpp`.
6. Sobre el nombre del proyecto, botón derecho **Set As Main Project** o desde el menú principal **Run – Set Main Project** y seleccionar este proyecto.
7. Con esto se puede ejecutar el proyecto normalmente. Obviamente, no hace nada porque está vacío y no se llama a ninguna función. A partir de aquí empezaría el desarrollo del proyecto, el cual cada programador seguirá un orden determinado y, probablemente, diferente a todos los demás, hasta que la aplicación esté terminada. En ese momento habrá que probarla, para ver que todo funciona como se espera, en un proceso que, de nuevo, dependerá de cada programador en concreto.



### 3.1.3. Configurar el entorno de testeo

Sin embargo, si se sigue la metodología TDD, este proceso es bien diferente, y se lleva a cabo de manera ordenada y sistemática.

1. En la vista lógica del proyecto, **Test Files – New Test Folder** y le ponemos un nombre, por ejemplo **TestsMyVector**.
2. **Test Files – Propiedades**.
  - En **C++ Compiler – Preprocessor Definitions** Añadir **\_\_INTEGRATION\_\_**
  - En **Linker – Libraries ... – Add Project** Añadir la carpeta principal del proyecto **GTest**. En **Additional Options ...** añadir **-pthread**.
3. **TestsMyVector – Add Existing Item ...** Añadir los tres ficheros **.cpp** que colocamos en la carpeta de tests **01\_Unit\_MyVector.cpp** – **02\_Unit\_MyVector.cpp** – **10\_Integration\_MyVector.cpp**.
4. Y ya estamos en posición de comenzar las pruebas, sí, con el proyecto aún sin implementar, TDD nos irá guiando. La primera opción es ejecutar los tests desde los controles de NetBeans, por ejemplo
  - a) **Run – Test Project**
  - b) Sobre la carpeta de tests **Tests folder** – Botón derecho y **Test**
  - c) Desde una terminal del proyecto, ejecutar **make test**.
5. Definir el entorno **Release**.
  - a) **Configuraciones**. Duplicar la configuración **Debug** y llamarla **Release**.
  - b) **C++ Compiler. Preprocessor Definitions**. Añadir **\_\_RELEASE\_\_**. Poner “Development mode” en **Release**.



## 4. Extensión de GoogleTest para la definición de tests de integración: MPTests

Esta sección describe las macros que se han definido en `MPTests.h` para cubrir algunas deficiencias de GoogleTest respecto a una asignatura como Metodología de la Programación.

### 4.1. Saltar un test

GoogleTest pasa siempre todos los tests que hayamos definido. Así, si tenemos un conjunto de tests muy amplio y falla el primero, esto no hace que GoogleTest se interrumpa, sino que sigue ejecutando todos los tests, con la consiguiente secuencia de fallos encadenados.

Por ejemplo, si nuestro constructor de la clase `Number` no estuviese bien definido y tuviésemos otro test adicional, el fallo del primero se acumularía al fallo del segundo (ver Figura 5).

```
TEST(MAIN, MiniClass_get) {
    Number n;
    ASSERT_EQ(n.get(), 0) << "Una instancia recién creada de Number, siempre devuelve el número 0" << endl;
}

TEST(Number, Number_increase) {
    Number a;           // Se inicializa a 0
    a.increase();       // Se incrementa en 1
    ASSERT_EQ(a.get(), 1) << "Si incrementamos una instancia recién creada de Number, el resultado debe ser 1" << endl;
}
```

Esto, en sí, no es un problema, pero sí es molesto, porque difumina el test que falla y se puede perder un poco el foco. Para ello, se han definido aserciones personalizadas en `MPTest` que, en cuanto falla un test, se salta (`SKIP_`) todos los demás y no genera más mensajes de error, permitiendo enfocar mejor el único tests que ha fallado.

```
TEST(MAIN, MiniClass_get) {
    Number n;
    SKIP_ASSERT_EQ_R(n.get(), 0) << "Una instancia recién creada de Number, siempre devuelve el número 0" << endl;
}
```

<code>SKIP_ASSERT_EQ_R(expr1, expr2)</code>	Las dos expresiones deben ser iguales
<code>SKIP_ASSERT_NE_R(expr1, expr2)</code>	Las dos expresiones deben ser diferentes
<code>SKIP_ASSERT_TRUE_R(expr1)</code>	La expresión debe ser verdadera
<code>SKIP_ASSERT_FALSE_R(expr1)</code>	La expresión debe ser falsa
<code>SKIP_ASSERT_STREQ_R(char*1, char*2)</code>	Las dos cadenas deben ser iguales
<code>SKIP_ASSERT_STNE_R(char*1, char*2)</code>	Las dos cadenas deben ser iguales

Cuadro 2: Algunas de las aserciones más frecuentes de GoogleTest



```
[=====] Running 9 tests from 3 test suites.
[-----] 4 tests from Level_1
[ RUN      ] Level_1.UnitNumero_get
tests/01.UnitNumero.cpp:14: Failure
Expected equality of these values:
  0
  n.get()
    Which is: 1
Una instancia recién creada de Numero,
siempre devuelve el número 0
[ FAILED ] Level_1.UnitNumero_get (0 ms)
[ RUN      ] Level_1.UnitNumero_Constructor
tests/01.UnitNumero.cpp:26: Skipped
[ SKIPPED ] Level_1.UnitNumero_Constructor (0 ms)
[ RUN      ] Level_1.UnitNumero_add
tests/01.UnitNumero.cpp:50: Skipped
[ SKIPPED ] Level_1.UnitNumero_add (0 ms)
[ RUN      ] Level_1.IntegrationA
tests/10.Integration_STATIC.cpp:13: Skipped
[ SKIPPED ] Level_1.IntegrationA (0 ms)
[-----] 4 tests from Level_1 (0 ms total)
[-----] 3 tests from Level_2
[ RUN      ] Level_2.UnitNumero_increase
tests/01.UnitNumero.cpp:21: Skipped
[ SKIPPED ] Level_2.UnitNumero_increase (0 ms)
[ RUN      ] Level_2.UnitNumero_isEven
tests/01.UnitNumero.cpp:59: Skipped
[ SKIPPED ] Level_2.UnitNumero_isEven (1 ms)
[ RUN      ] Level_2.IntegrationC
tests/10.Integration_STATIC.cpp:25: Skipped
[ SKIPPED ] Level_2.IntegrationC (0 ms)
[-----] 3 tests from Level_2 (1 ms total)
[-----] 2 tests from Level_3
[ RUN      ] Level_3.UnitNumero_increase_loop
tests/01.UnitNumero.cpp:37: Skipped
[ SKIPPED ] Level_3.UnitNumero_increase_loop (0 ms)
[ RUN      ] Level_3.IntegrationZ
tests/10.Integration_STATIC.cpp:33: Skipped
[ SKIPPED ] Level_3.IntegrationZ (0 ms)
[-----] 2 tests from Level_3 (0 ms total)
[-----] Global test environment tear-down
[=====] 9 tests from 3 test suites ran. (1 ms)
[ PASSED ] 0 tests.
[ SKIPPED ] 8 tests, listed below:
[ SKIPPED ] Level_1.UnitNumero_Constructor
[ SKIPPED ] Level_1.UnitNumero_add
[ SKIPPED ] Level_1.IntegrationA
[ SKIPPED ] Level_2.UnitNumero_increase
[ SKIPPED ] Level_2.UnitNumero_isEven
[ SKIPPED ] Level_2.IntegrationC
[ SKIPPED ] Level_3.UnitNumero_increase_loop
[ SKIPPED ] Level_3.IntegrationZ
[ FAILED ] 1 test, listed below:
[ FAILED ] Level_1.UnitNumero_get
1 FAILED TEST

[=====] Running 9 tests from 3 test suites.
[-----] 4 tests from Level_1
[ RUN      ] Level_1.UnitNumero_get
tests/01.UnitNumero.cpp:14: Failure
Expected equality of these values:
  0
  n.get()
    Which is: 1
Una instancia recién creada de UnitNumero,
siempre devuelve el número 0
[ FAILED ] Level_1.UnitNumero_get (0 ms)
[ RUN      ] Level_1.UnitNumero_Constructor
tests/01.UnitNumero.cpp:26: Failure
Expected equality of these values:
  "Number::[0]"
  n.inspect().c_str()
    Which is: "Number::[1]"
A newly created instance always gives "(0)" as output
[ FAILED ] Level_1.UnitNumero_Constructor (0 ms)
[ RUN      ] Level_1.UnitNumero_add
[ OK      ] Level_1.UnitNumero_add (1 ms)
[ RUN      ] Level_1.IntegrationA
tests/10.Integration_STATIC.cpp:21: Failure
Expected equality of these values:
  EXPECTED_OUTPUT
    Which is: "[a] Number::[0] [a] Number::[0]"
  [a] Number::[1] El resultado es impar: 5 [a] Number::[5]"
  REAL_OUTPUT
    Which is: "[a] Number::[1] [a] Number::[1]"
  [a] Number::[2] El resultado es par: 6 [a] Number::[6]"
[ FAILED ] Level_1.IntegrationA (6 ms)
[-----] 4 tests from Level_1 (7 ms total)
[-----] 3 tests from Level_2
[ RUN      ] Level_2.UnitNumero_increase
tests/01.UnitNumero.cpp:21: Failure
Expected equality of these values:
  1
  a.get()
    Which is: 2
Si incrementamos una instancia recién creada de
Numero, el resultado debe ser 1
[ FAILED ] Level_2.UnitNumero_increase (0 ms)
[ RUN      ] Level_2.UnitNumero_isEven
[ OK      ] Level_2.UnitNumero_isEven (0 ms)
[ RUN      ] Level_2.IntegrationC
tests/10.Integration_STATIC.cpp:28: Failure
Expected equality of these values:
  EXPECTED_OUTPUT
    Which is: "[a] Number::[0] [a] Number::[0]"
  [a] Number::[1] El resultado es impar: 5 [a] Number::[5]"
  REAL_OUTPUT
    Which is: "[a] Number::[1] [a] Number::[1]"
  [a] Number::[2] El resultado es par: 6 [a] Number::[6]"
La introducción de los valores 2 y 3 desde el teclado debe dar
5 como resultado
[ FAILED ] Level_2.IntegrationC (6 ms)
[-----] 3 tests from Level_2 (6 ms total)
[-----] 2 tests from Level_3
[ RUN      ] Level_3.UnitNumero_increase_loop
[ OK      ] Level_3.UnitNumero_increase_loop (1 ms)
[ RUN      ] Level_3.IntegrationZ
tests/10.Integration_STATIC.cpp:36: Failure
Expected equality of these values:
  EXPECTED_OUTPUT
    Which is: "[a] Number::[0] El resultado es par: 0"
  [a] Number::[0]"
  REAL_OUTPUT
    Which is: "[a] Number::[1] El resultado es impar: 1"
  [a] Number::[1]"
La introducción de los valores 0 y 0 desde el teclado debe
dar 0 como resultado
[ FAILED ] Level_3.IntegrationZ (9 ms)
[-----] 2 tests from Level_3 (10 ms total)
[=====] 9 tests from 3 test suites ran. (23 ms total)
[ PASSED ] 3 tests.
[ FAILED ] 6 tests, listed below:
[ FAILED ] Level_1.UnitNumero_get
[ FAILED ] Level_1.UnitNumero_Constructor
[ FAILED ] Level_1.IntegrationA
[ FAILED ] Level_2.UnitNumero_increase
[ FAILED ] Level_2.IntegrationC
[ FAILED ] Level_3.IntegrationZ
6 FAILED TESTS
```

Figura 5: Diferencias entre el uso de las aserciones `SKIP_ASSERT.*` (a la izquierda) y las habituales de GoogleTest `ASSERT.*` (a la derecha).

## 4.2. Tests de integración

Los tests unitarios se centran en comprobar que las unidades funcionales hacen lo que se espera que hagan, pero no comprueban el funcionamiento de `main()`. Esto es un problema muy abierto para el que existen múltiples plataformas, la mayoría de ellas demasiado complejas como para ser introducidas en el marco de una asignatura de primero de grado como es Metodología de la Programación. No obstante, dada la base de confianza que proporcionan los tests unitarios y, a partir de las experiencias previas en esta asignatura en DGIIM, se pueden realizar algunos tests de integración relativamente sencillos, que permiten ejecutar los programas completos, con su verdadero `main`, contra una serie de datos de entrada controlados por el profesor, y de los cuales se conoce la salida esperada. Como el siguiente programa.

```
int main() {
    Number a;
    int increm1, increm2;
    cout << "Introduzca dos incrementos consecutivos ";
    cin >> increm1 >> increm2;
    for (int i=0; i<increm1; a.increase(), i++);
    for (int i=0; i<increm2; a.increase(), i++);
    if (a.isEven()) {
        cout << "El resultado es par: " << a.get() << endl;
    } else {
        cout << "El resultado es impar: " << a.get() << endl;
    }
}

$$ Introduzca dos incrementos consecutivos 2 3
$$ El resultado es impar: 5
```

Para ello, se ejecuta el programa y se captura parte de su salida, marcando en esta salida, los checkpoints que se consideren necesarios y permitiendo la libertad del alumno para programar sus propias salidas, en los que se analiza, tanto la salida del programa, como el estado interno de cada dato del programa que pueda ser relevante. Finalmente, el test de integración comprueba que la salida esperada coincide con la salida detectada real en esos checkpoints.

### 4.2.1. Definir el entorno de ejecución controlado para testear la aplicación completa

La macro `DEF_EXECUTION_ENVIRONMENT(<LABEL>)` permite crear un entorno controlado para la ejecución de la aplicación, en la que se puede simular la redirección de entrada y la de salida para comprobar distintos casos de ejecución. Cada caso se programaría en un test diferente. la etiqueta debería coincidir con el nombre del test (segundo parámetro en la llamada a `TEST(C,A)`).

La macro `FROM_KEYBOARD()` permite definir una entrada simulada desde el teclado, la cual, al llamar al binario con la macro `CALL_FROM_KEYBOARD()` simula esa entrada desde `cin` al ejecutar el binario.

```
TEST(MAIN, IntegrationA) {
    DEF_EXECUTION_ENVIRONMENT(IntegrationA);
    FROM_KEYBOARD("2 3");
    CALL_FROM_KEYBOARD(" ");
}
```

En caso de que la simulación de los datos de entrada desde teclado sean muy largas como para ponerlas en una línea con `FROM_KEYBOARD()` se podría grabar esta secuencia de datos en un fichero con extensión `input` en la carpeta

./tests/validation/. Nótese que el nombre del fichero debe coincidir con el nombre del test, en este caso “IntegrationA”;

```

$ echo "2 3" > ./tests/validation/IntegrationA.input

TEST(MAIN, IntegrationA) {
    DEF_EXECUTION_ENVIRONMENT(IntegrationA);
    CALL_FROM_FILE(IntegrationA, "");
}

```

Esto hace que se ejecute el programa completamente, pero su salida, ni aparece en la pantalla durante el test, ni podemos comprobar que es la esperada.

#### 4.2.2. Definiendo checkpoints en la salida

El alumno puede programar su salida con `cout` como individualmente prefiera, pero puede incluir checkpoints para validar los tests de integración mediante la macro `CVAL`. Cuando se está en testeo, esta macro se expande a `cerr`, en otro caso, se expande a `cout`, lo que permite capturar la salida del programa, sólo en aquellos puntos que se desee.

```

int main() {
    Number a;

    int increm1, increm2;
    cout << "Introduzca dos incrementos consecutivos ";
    cin >> increm1 >> increm2;
    for (int i=0; i<increm1; a.increase(), i++);
    for (int i=0; i<increm2; a.increase(), i++);
    if (a.isEven()) {
        CVAL << "El resultado es par: " << a.get() << endl;
    } else {
        CVAL << "El resultado es impar: " << a.get() << endl;
    }
}

```

```

TEST(MAIN, IntegrationA) {
    DEF_EXECUTION_ENVIRONMENT(IntegrationA);
    FROM_KEYBOARD("2 3");
    EXPECTED_TEXT_OUTPUT("El resultado es impar: 5");
    CALL_FROM_KEYBOARD("") << "La introducción de los valores 2 y 3 desde el teclado debe dar 5 como resultado" << endl;
}

TEST(MAIN, IntegrationB) {
    DEF_EXECUTION_ENVIRONMENT(IntegrationB);
    FROM_KEYBOARD("2 2");
    EXPECTED_TEXT_OUTPUT("El resultado es par: 4");
    CALL_FROM_KEYBOARD("") << "La introducción de los valores 2 y 2 desde el teclado debe dar 4 como resultado" << endl;
}

```

Para ello, la macro `EXPECTED_TEXT_OUTPUT` define cómo debería de haber sido la salida en el caso de que fuese correcta, eliminando espacios duplicados y saltos de línea. Esta salida esperada se contrasta contra la salida real en la última llamada `ASSERT_STREQ` que comprueba si ambas salidas, especificadas como secuencias de caracteres, coinciden o no.

```

[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from MAIN
[ RUN    ] MAIN.IntegrationA
[ OK     ] MAIN.IntegrationA (7 ms)
[ RUN    ] MAIN.IntegrationB
[ OK     ] MAIN.IntegrationB (6 ms)
[-----] 2 tests from MAIN (13 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (13 ms total)
[ PASSED ] 2 tests.

RUN FINISHED: exit value 0; real time: 20ms; user: 0ms; system: 10ms

```

Hay que tener mucho cuidado con estos tests de integración pues hacen una comparación carácter a carácter de forma estricta por lo que conviene saber exactamente cómo se produce la salida a través de CVAL. En cualquier caso, cuando la salida esperada difiere de la salida real obtenida, MPTTest muestra la salida tal y como debía haber sido, y marca las diferencias con la sintaxis del comando `wdiff`<sup>7</sup>. Por ejemplo, si la salida esperada debería haber sido

"A B C D E F G"

entonces las siguientes salidas producirían las siguientes diferencias

Salida real	wdiff	En esta
" "	"[-A B C D E F G-]"	
"A B C D "	"A B C D [-E F G-]"	
"A B C D F G"	"A B C D [-E-] F G"	
"A B C D H F G"	"A B C D [-E-] {+H+} F G"	
"1 2 3 4 5 6 7"	"[-A B C D E F G-]{+1 2 3 4 5 6 7+}"	

sintaxis, las palabras que deberían estar en la salida, pero no lo están, se marcan con `[- <palabra> -]` y las que sí están en la salida, pero sobran, se marcan con `{+ <palabra> +}`

Por ejemplo, si el programa anterior estuviese mal y tuviese los casos par e impar intercambiados por error

```
int main() {
    Number a;

    int increm1, increm2;
    cout << "Introduzca dos incrementos consecutivos ";
    cin >> increm1 >> increm2;
    for (int i=0; i<increm1; a.increase(), i++);
    for (int i=0; i<increm2; a.increase(), i++);
    if (!a.isEven()) { // Error: está al revés
        CVAL << "El resultado es par: " << a.get() << endl;
    } else {
        CVAL << "El resultado es impar: " << a.get() << endl;
    }
}
```

Entonces la salida del test hubiese sido también diferente, se puede ver que el valor calculado es correcto, pero las palabras par e impar aparecen intercambiadas.

<sup>7</sup>Para ello debe de estar instalado el programa `wdiff`

```

[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from MAIN
[ RUN      ] MAIN.IntegrationA
tests/10.Integration_STATIC_Latex.cpp:34: Failure
Expected equality of these values:
""
_wdiff.c_str()
Which is: "El resultado es [-impar:-] (+par:+) 5"
La introducción de los valores 2 y 3 desde el teclado debe dar 5 como resultado

[  FAILED  ] MAIN.IntegrationA (13 ms)
[ RUN      ] MAIN.IntegrationB
tests/10.Integration_STATIC_Latex.cpp:41: Failure
Expected equality of these values:
""
_wdiff.c_str()
Which is: "El resultado es [-par:-] (+impar:+) 4"
La introducción de los valores 2 y 2 desde el teclado debe dar 4 como resultado

[  FAILED  ] MAIN.IntegrationB (14 ms)
[-----] 2 tests from MAIN (27 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (27 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 2 tests, listed below:
[  FAILED  ] MAIN.IntegrationA
[  FAILED  ] MAIN.IntegrationB

```

### 4.2.3. Aplicaciones embucladas

Es muy frecuente que, debido a errores de programación, se creen ciclos infinitos en la ejecución, lo que llevaría a un bloqueo de todos los tests. Para resolver este problema, todas las aplicaciones se ejecutan con un time-out, por defecto en 30 segundos, transcurridos los cuales, se interrumpe la aplicación y el test se da por fallado. En caso de aplicaciones que necesiten más tiempo para una ejecución normal, se puede personalizar este time-out con la macro `SET_TIMEOUT_S` que especifica un tiempo de ejecución máximo en segundos.

```

TEST(UnitNumber, IntegrationFrozen) {
    DEF_EXECUTION_ENVIRONMENT(IntegrationFrozen);
    FROM_KEYBOARD("2_2147483647");
    SET_TIMEOUT_S(3);
    EXPECTED_TEXT_OUTPUT("El resultado es par: 4");
    CALL_FROM_KEYBOARD("**)<<"La introducción de los valores 2 y 2 desde el teclado debe dar 4 como resultado "<<endl;
}

```

### 4.2.4. Validando el contenido de ciertas clases

Además, se pueden aprovechar los checkpoints de salida controlada para serializar ciertas clases y capturarlas durante la ejecución, en los puntos que se estime necesario. Para ello se usa la combinación de la macro `REPORT_DATA(x)` (Figura 6) que vuelca en CVAL la salida del método `report_data()`, el cual debe de estar instanciado en la clase a la que pertenece el objeto `x`.

Cuando no está en testeo, esta última macro desaparece y no hace nada para no interferir con la salida real de la aplicación (Figura 7). En ciertas clases cuya serialización sea muy extensa, el método `inspect()` puede generar un hash, un código encriptado, que representa dicha serialización, pudiendo compararse la serialización esperada con la real sin más que comparar estos códigos.

### 4.2.5. Detectando problemas de memoria dinámica

Otra de las propiedades de la extensión `MPTTest` es que permite pasar un control de memoria dinámica, integrado y homogéneo, mediante el uso del

```
class Number {
private:
    int data;
public:
    Number() { data = 0; }
    int get() const { return data; }
    int increase() {data++;}
    void add(Number n) {data += n.get();}
    bool isEven() {return data %2 == 0;}
    // Serialización de la clase para inspeccionar su contenido en determinados puntos
    // del programa
    std::string inspect() {return "{Number}::[" +std::to_string(data)+" "];}
};
```

(a)

```
int main() {
    Number a;
    REPORT_DATA(a);      // Chekpoint inicial
    int increm1, increm2;
    cout << "Introduzca dos incrementos consecutivos ";
    cin >> increm1>>increm2;
    for (int i=0;i<increm1;a.increase(), i++) REPORT_DATA(a);    // Chekpoint continuo
    for (int i=0;i<increm2;a.increase(), i++);
    if (a.isEven()) {
        CVAL << "El resultado es par: " << a.get() << endl;
    } else {
        CVAL << "El resultado es impar: " << a.get() << endl;
    }
    REPORT_DATA(a);      // Chekpoint final
}
```

(b)

```
[a] {Number}::[0]
Introduzca dos incrementos consecutivos 4 1
[a] {Number}::[0]
[a] {Number}::[1]
[a] {Number}::[2]
[a] {Number}::[3]
El resultado es impar: 5
[a] {Number}::[5]
}
```

(c)

```
TEST(UnitNumber, IntegrationA) {
    DEF_EXECUTION_ENVIRONMENT(IntegrationA);
    EXPECTED_OUTPUT(
[a] {Number}::[0]
[a] {Number}::[0]
[a] {Number}::[1]
El resultado es impar: 5
[a] {Number}::[5]);
    CALL_FROM_FILE("");
}
```

(d)

Figura 6: (a) Fichero `Number.h` con el método `inspect()` para examinar su contenido detallado durante la ejecución, en caso de que fuese necesario. (b) Función `main()` en la que se han puesto varios puntos de control para examinar en detalle la clase `Number` conforme avanza el programa. (c) Salida del programa que muestra la evolución del interior de la clase `Number` a lo largo de la ejecución del programa. (d) Test de integración que comprueba que, tanto la salida como la evolución de la clase son correctas.

```
Introduzca dos incrementos consecutivos 4 1
El resultado es impar: 5
```

Figura 7: Salida del mismo programa de la Figura 6 en el modo `Release`, en el que se puede apreciar que las salidas de validación han desaparecido

```
TEST(UnitNumber, IntegrationC) {
    DEF_EXECUTION_ENVIRONMENT(IntegrationC);
    FROM_KEYBOARD("23");
    USE_MEMORY_LEAKS(__VALGRIND__);
    EXPECTED_TEXT_OUTPUT( "[a]{Number}::[0][a]{Number}::[1][a]{Number}::[2]El
    resultado es impar: 5[a]{Number}::[5]");
    CALL_FROM_KEYBOARD(" ") << "La introducción de los valores 2 y 3 desde el teclado debe
    dar 5 como resultado" << endl;
}
```

Figura 8: Incorporación de chequeo de memoria dinámica en un test, pudiendo utilizar, si están instalados, los paquetes `__VALGRIND__` o `__DRMEMORY__`

entorno **Valgrind**<sup>8</sup> o **DrMemory**<sup>9</sup>

Para ello se dispone de la macro `USE_MEMORY_LEAKS(__VALGRIND__)` o `USE_MEMORY_LEAKS(__DRMEMORY__)` que están integradas en el entorno de ejecución y proporcionan la misma salida en GoogleTest, para ambas herramientas, de los errores más comunes. En este caso, la aplicación es suficientemente sencilla como para no producir errores en un test básico como el mostrado en la Figura 9.

La inclusión del chequeo de memoria dinámica es muy sencilla. Tan solo hay que solicitarla y esperar al resultado. En primer lugar se pasa el chequeo de memoria, porque es lo más prioritario. Si este falla, todo el test falla, en otro caso, se pasa el test en sí mismo con la validación de datos. El resultado aparece integrado en la salida de GoogleTest como un test más que se identifica como “MEMCHECK” e indica el paquete con el que se ha realizado.

```
[=====] Running 9 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 9 tests from UnitNumber
[ RUN      ] UnitNumber.UnitNumber_get
[ OK       ] UnitNumber.UnitNumber_get (0 ms)
[ RUN      ] UnitNumber.UnitNumber_increase
[ OK       ] UnitNumber.UnitNumber_increase (1 ms)
[ RUN      ] UnitNumber.UnitNumber_Constructor
[ OK       ] UnitNumber.UnitNumber_Constructor (0 ms)
[ RUN      ] UnitNumber.IntegrationC
[ MEMCHECK ] IntegrationC-valgrind
[ OK       ] IntegrationC-valgrind
[ OK       ] UnitNumber.IntegrationC (1024 ms)
```

<sup>8</sup>Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

The Valgrind distribution currently includes seven production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and two different heap profilers. It also includes an experimental SimPoint basic block vector generator. It runs on the following platforms: X86/Linux, AMD64/Linux. <https://www.valgrind.org/>

<sup>9</sup>Dr. Memory is a memory monitoring tool capable of identifying memory-related programming errors such as accesses of uninitialized memory, accesses to unaddressable memory (including outside of allocated heap units and heap underflow and overflow), accesses to freed memory, double frees, memory leaks, and (on Windows) handle leaks, GDI API usage errors, and accesses to un-reserved thread local storage slots. Dr. Memory operates on unmodified application binaries running on Windows, Linux, Mac, or Android



Veamos qué ocurre si se producen errores de memoria poniendo un main más complejo. El siguiente `main()` produce casi todos los tipos de error que se van a considerar:

**NO\_INIT** Uso de memoria no inicializada.

**BAD\_READ** Acceso de lectura a una posición de memoria no permitida.

**BAD\_WRITE** Acceso de escritura a una posición de memoria no permitida.

**MEM\_LEAK** Pérdida de memoria debido a que no se ha liberado antes de terminar el programa.

**BAD\_FREE** Uso del operador inadecuado para liberar la memoria.

**CORE\_DUMP** Violación de segmento.

```
int main() {
    Number *a;
    a = new Number [SIZE];

    REPORT_DATA((*a)); /// @brief Checkpoint
    inicial
    int increm1, increm2, increm3;
    cout << "Introduzca dos incrementos consecutivos";
    cin >> increm1 >> increm2;
    for (int i = 0; i < increm1; a[1].
    increase(), i++) REPORT_DATA((*a)); ///
    @brief Checkpoint continuo
    for (int i = 0; i < increm3; a[0].
    increase(), i++) {
        if (a[0].isEven()) {
            CVAL << "El resultado es par: " << a
            [0].get() << endl;
        } else {
            CVAL << "El resultado es impar: " <<
            a[0].get() << endl;
        }
    }
    REPORT_DATA((*a)); /// @brief Checkpoint
    final
    //delete [] a;
}
```

```
Running 9 tests from 2 test suites.
Global test environment set-up.
6 tests from UnitNumber
PASS: UnitNumber.UnitNumber_get
PASS: UnitNumber.UnitNumber_get (0 ms)
PASS: UnitNumber.UnitNumber_increase
PASS: UnitNumber.UnitNumber_increase (0 ms)
PASS: UnitNumber.UnitNumber_Constructor
PASS: UnitNumber.UnitNumber_Constructor (0 ms)
PASS: UnitNumber.UnitNumber_increase_loop
PASS: UnitNumber.UnitNumber_increase_loop (0 ms)
PASS: UnitNumber.UnitNumber_add
PASS: UnitNumber.UnitNumber_add (0 ms)
PASS: UnitNumber.UnitNumber_isEven
PASS: UnitNumber.UnitNumber_isEven (0 ms)
6 tests from UnitNumber (2 ms total)

3 tests from IntegrationNumber
PASS: IntegrationNumber.IntegrationA
[ MEMCHECK ] IntegrationA_valgrind
[ BAD_READ ] (main_Heap.cpp:21) (main_Heap.cpp:49) (main_Heap.cpp:43)
[ BAD_WRITE ] (main_Heap.cpp:21) (main_Heap.cpp:49) (main_Heap.cpp:43)
[ NO_INIT ] (main_Heap.cpp:59)
[ MEM_LEAK ] (main_Heap.cpp:43)
src/main_Heap.cpp:126: Failure
Failed
[ FAILED ] IntegrationNumber.IntegrationA (1085 ms)
PASS: IntegrationNumber.IntegrationC
src/main_Heap.cpp:130: Skipped
[ SKIPPED ] IntegrationNumber.IntegrationC (0 ms)
PASS: IntegrationNumber.IntegrationZ
src/main_Heap.cpp:139: Skipped
[ SKIPPED ] IntegrationNumber.IntegrationZ (0 ms)
3 tests from IntegrationNumber (1085 ms total)

Global test environment tear-down
9 tests from 2 test suites ran. (1087 ms total)
6 tests.
[ SKIPPED ] 2 tests, listed below:
[ SKIPPED ] IntegrationNumber.IntegrationC
[ SKIPPED ] IntegrationNumber.IntegrationZ
[ FAILED ] 1 test, listed below:
[ FAILED ] IntegrationNumber.IntegrationA
1 FAILED TEST
```

Figura 9: Izquierda: Código con varios errores de gestión de memoria. Derecha: Informe de errores incrustado en GoogleTest mostrando el error y las líneas de código afectadas.

En este caso, se puede ver en la Figura 9 el informe de GoogleTest al que estamos acostumbrados, arriba a la derecha, el cual sigue el mismo formato con ambas herramientas de análisis de memoria, indicando, siempre que sea posible, las líneas de código en las que se producen estos errores.

#### 4.2.6. Paso de parámetros a main

Para terminar, no sólo se pueden redireccionar la entrada y la salida desde el mismo entorno de Test para comprobar que la aplicación funciona como se esperaba, sino que se le pueden pasar argumentos a main desde la misma llamada del test, para completar todas las posibilidades de testeo de integración que cubre MPTTest y que son las que se desarrollan en las asignaturas Fundamentos de la Programación y Metodología de la Programación.

```
TEST(INTEGRATION, CALL_ERROR_BAD_OPTION) {
    DEF_EXECUTION_ENVIRONMENT(CALL_ERROR_BAD_OPTION);
    EXPECTED_TEXT_OUTPUT( "Error in call" );
    CALL( "-lES-r2020-k" ) << "An unexpected parameter in the command line must produce
    \ "Error in call \ " << endl;
}
```

### 4.3. Reportes de tests

Además de todo lo anterior, la extensión MPTTest genera varios informes.

#### 4.3.1. Reporte inicial de los tests que hay preparados

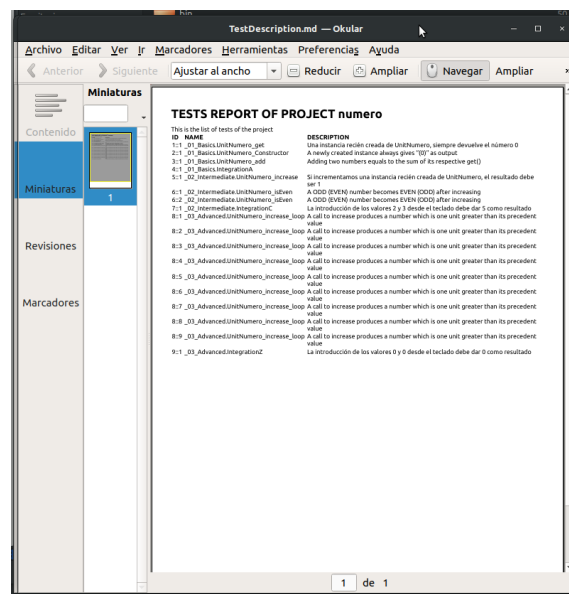


Figura 10: Informe preliminar de tests que se van a pasar.

La extensión **MPTtest** puede generar un reporte inicial que sirve de guía durante el proceso de testeo. Este informe contiene todos los tests que haya activos en NetBeans, con la misma numeración que saldrán en la pantalla y que el reporte anterior, a modo de guía para el testeo, pero antes de pasar los tests. Este test hay que generarlo sólo una vez, y se hace manualmente desde una consola del proyecto en la que se invoca a los tests.

```
make clean
make CXXFLAGS=-D__REPORT_ALL_TESTS__ test
make clean
```

Este reporte se genera en

`./doc/markdown/TestDescription.md`

se puede visualizar con un visor de PDFs como okular (ver Figura 10).

#### 4.3.2. Reporte del resultado de pasar los tests

El primero es un informe en formato Markdown que describe el resultado auditado de los tests, por si hubiese que justificarlo. Este informe se encuentra en la carpeta

`./tests/output/TestReport.md`

y se puede visualizar con un visor de PDFs como *okular* (ver Figura 11).

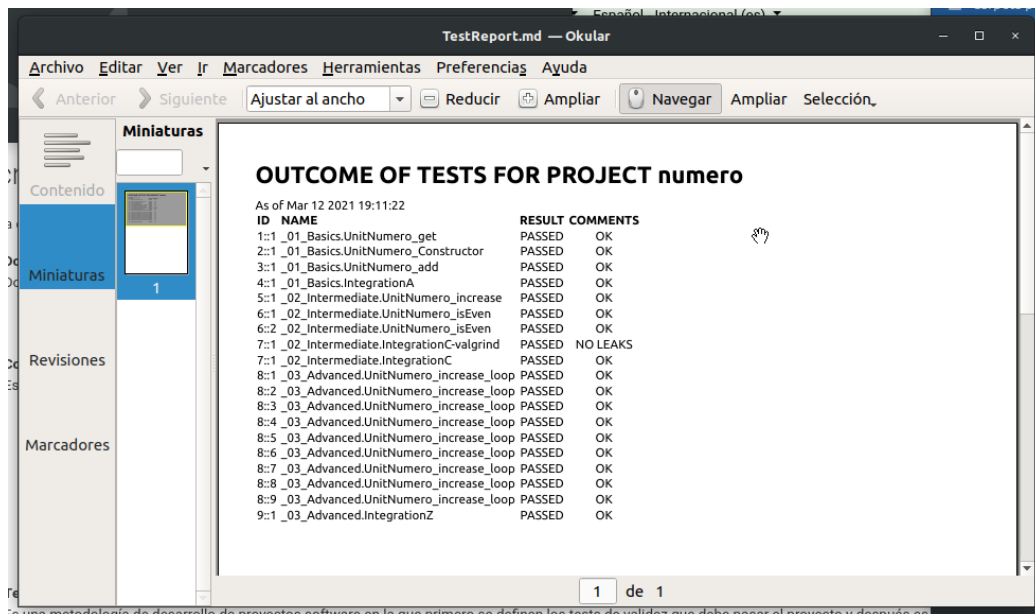


Figura 11: Reporte de tests pasados. Una vez terminado el proceso de testeo, se genera un informe de auditoría de tests situado en `./tests/output/TestReport.md`. La primera columna muestra dos números, el primero es el ordinal del test, coincidiendo con el ordinal de Googletest (ver Figura 5). Cada test puede tener varios chequeos en su interior, para ello, el segundo índice hace referencia al orden en el que aparece este chequeo.

Este reporte muestra todos los tests, tanto unitarios como de integración, que se han preparado y su resultado. Obsérvese que el test de la Figura 8, cuyo nombre es *IntegrationC* aparece varias veces, una de ellas, la primera, es el chequeo de memoria (PASSED - NO LEAKS), y la otra es el test en sí (PASSED - OK).

La Figura 12 muestra sendos reportes para los problemas de memoria mostrados en la Figura 9. A la izquierda el informe detallado que genera *MPTTest* cuando se selecciona *Valgrind* y a la derecha el informe de *Dr. Memory*. Se puede ver que, aunque el informe en pantalla integrado con *GoogleTest* se ha unificado por igual con ambas herramientas (Figura 9), el informe particular puede variar, tanto en los mensajes propios de cada uno como en la detección.

## 5. Propiedades de TDD

Algunas propiedades de TDD que conviene subrayar son las siguientes:

1. Corrección. Los tests deben de ser correctos. Es código, así que el mismo test podría contener errores, lo que llevaría a una peligrosa validación de las unidades funcionales.
2. Completitud. El conjunto de tests unitarios debe ser completo o lo más completo posible, para cubrir todas las posibles casuísticas.

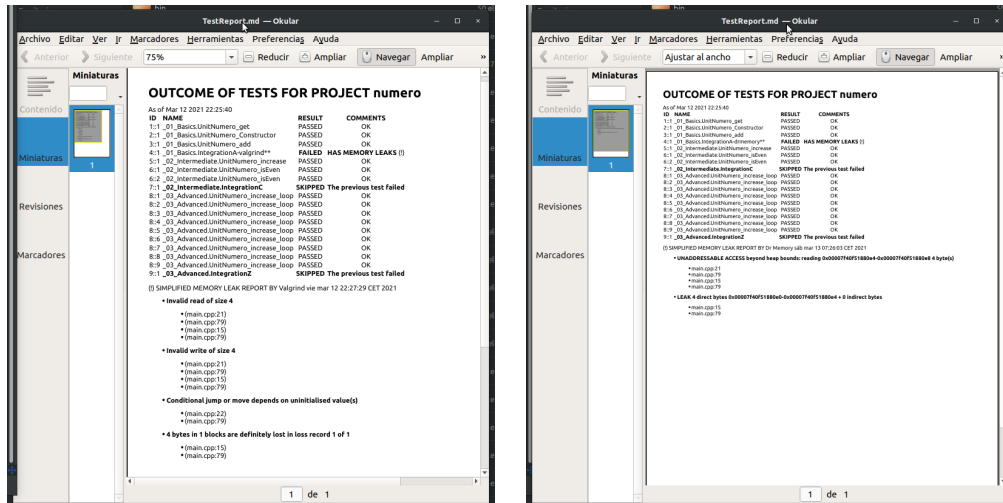


Figura 12: Informes de error generados por MPTTest para el caso de la Figura 9, a la izquierda el de Valgrind y a la derecha el de Dr. Memory.

3. Legibilidad. Tanto el test como su resultado, sobre todo en caso de fallo, debe ser lo más claro posible.
4. Repetibilidad. En caso de repetir varias veces un mismo test con los mismos datos, debe producir el mismo resultado.
5. Agilidad. Cada test debe de ejecutarse en el mínimo tiempo posible y se debe ejecutar al alcance del mínimo número de clics.



## 6. (git) About GoogleTest

### OSS Builds Status

#### 6.0.1. Announcements

**Release 1.10.x** Release 1.10.x is now available.

#### Coming Soon

- Post 1.10.x googletest will follow Abseil Live at Head philosophy
- We are also planning to take a dependency on Abseil.

### 6.1. Welcome to GoogleTest, Google's C++ test framework!

This repository is a merger of the formerly separate GoogleTest and GoogleMock projects. These were so closely related that it makes sense to maintain and release them together.

#### 6.1.1. Getting Started

The information for **GoogleTest** is available in the GoogleTest Primer documentation.

**GoogleMock** is an extension to GoogleTest for writing and using C++ mock classes. See the separate GoogleMock documentation.

More detailed documentation for googletest is in its interior googletest/README.md file.

### 6.2. Features

- An xUnit test framework.
- Test discovery.
- A rich set of assertions.
- User-defined assertions.
- Death tests.
- Fatal and non-fatal failures.
- Value-parameterized tests.
- Type-parameterized tests.
- Various options for running the tests.
- XML test report generation.

### 6.3. Supported Platforms

GoogleTest requires a codebase and compiler compliant with the C++11 standard or newer.

The GoogleTest code is officially supported on the following platforms. Operating systems or tools not listed below are community-supported. For



community-supported platforms, patches that do not complicate the code may be considered.

If you notice any problems on your platform, please file an issue on the GoogleTest GitHub Issue Tracker. Pull requests containing fixes are welcome!

### 6.3.1. Operating Systems

- Linux
- macOS
- Windows

### 6.3.2. Compilers

- gcc 5.0+
- clang 5.0+
- MSVC 2015+

**macOS users:** Xcode 9.3+ provides clang 5.0+.

### 6.3.3. Build Systems

- Bazel
- CMake

**Note:** Bazel is the build system used by the team internally and in tests. CMake is supported on a best-effort basis and by the community.

## 6.4. Who Is Using GoogleTest?

In addition to many internal projects at Google, GoogleTest is also used by the following notable projects:

- The Chromium projects (behind the Chrome browser and Chrome OS).
- The LLVM compiler.
- Protocol Buffers, Google's data interchange format.
- The OpenCV computer vision library.

## 6.5. Related Open Source Projects

GTest Runner is a Qt5 based automated test-runner and Graphical User Interface with powerful features for Windows and Linux platforms.

GoogleTest UI is a test runner that runs your test binary, allows you to track its progress via a progress bar, and displays a list of test failures. Clicking on one shows failure text. Google Test UI is written in C#.

GTest TAP Listener is an event listener for GoogleTest that implements the TAP protocol for test result output. If your test runner understands TAP, you may find it useful.

gtest-parallel is a test runner that runs tests from your binary in parallel to provide significant speed-up.



GoogleTest Adapter is a VS Code extension allowing to view GoogleTest in a tree view, and run/debug your tests.

C++ TestMate is a VS Code extension allowing to view GoogleTest in a tree view, and run/debug your tests.

Cornichon is a small Gherkin DSL parser that generates stub code for GoogleTest.

## **6.6. Contributing Changes**

Please read `CONTRIBUTING.md` for details on how to contribute to this project.

Happy testing!

## 7. (git) GoogleTest Primer

### 7.1. Introduction: Why googletest?

*googletest* helps you write better C++ tests.

*googletest* is a testing framework developed by the Testing Technology team with Google's specific requirements and constraints in mind. Whether you work on Linux, Windows, or a Mac, if you write C++ code, *googletest* can help you. And it supports *any* kind of tests, not just unit tests.

So what makes a good test, and how does *googletest* fit in? We believe:

1. Tests should be *independent* and *repeatable*. It's a pain to debug a test that succeeds or fails as a result of other tests. *googletest* isolates the tests by running each of them on a different object. When a test fails, *googletest* allows you to run it in isolation for quick debugging.
2. Tests should be well *organized* and reflect the structure of the tested code. *googletest* groups related tests into test suites that can share data and subroutines. This common pattern is easy to recognize and makes tests easy to maintain. Such consistency is especially helpful when people switch projects and start to work on a new code base.
3. Tests should be *portable* and *reusable*. Google has a lot of code that is platform-neutral; its tests should also be platform-neutral. *googletest* works on different OSes, with different compilers, with or without exceptions, so *googletest* tests can work with a variety of configurations.
4. When tests fail, they should provide as much *information* about the problem as possible. *googletest* doesn't stop at the first test failure. Instead, it only stops the current test and continues with the next. You can also set up tests that report non-fatal failures after which the current test continues. Thus, you can detect and fix multiple bugs in a single run-edit-compile cycle.
5. The testing framework should liberate test writers from housekeeping chores and let them focus on the test *content*. *googletest* automatically keeps track of all tests defined, and doesn't require the user to enumerate them in order to run them.
6. Tests should be *fast*. With *googletest*, you can reuse shared resources across tests and pay for the set-up/tear-down only once, without making tests depend on each other.

Since *googletest* is based on the popular xUnit architecture, you'll feel right at home if you've used JUnit or PyUnit before. If not, it will take you about 10 minutes to learn the basics and get started. So let's go!

### 7.2. Beware of the nomenclature

*Note:* There might be some confusion arising from different definitions of the terms *Test*, *Test Case* and *Test Suite*, so beware of misunderstanding these.

Historically, *googletest* started to use the term *Test Case* for grouping related tests, whereas current publications, including International Software Testing Qualifications Board (ISTQB) materials and various textbooks on software



quality, use the term *Test Suite* for this.

The related term *Test*, as it is used in googletest, corresponds to the term *Test Case* of ISTQB and others.

The term *Test* is commonly of broad enough sense, including ISTQB's definition of *Test Case*, so it's not much of a problem here. But the term *Test Case* as was used in Google Test is of contradictory sense and thus confusing.

googletest recently started replacing the term *Test Case* with *Test Suite*. The preferred API is *TestSuite*. The older *TestCase* API is being slowly deprecated and refactored away.

So please be aware of the different definitions of the terms:

Meaning	googletest Term	ISTQB Term
Exercise a particular program path with specific input values and verify the results	TEST()	Test Case

### 7.3. Basic Concepts

When using googletest, you start by writing *assertions*, which are statements that check whether a condition is true. An assertion's result can be *success*, *nonfatal failure*, or *fatal failure*. If a fatal failure occurs, it aborts the current function; otherwise the program continues normally.

*Tests* use assertions to verify the tested code's behavior. If a test crashes or has a failed assertion, then it *fails*; otherwise it *succeeds*.

A *test suite* contains one or many tests. You should group your tests into test suites that reflect the structure of the tested code. When multiple tests in a test suite need to share common objects and subroutines, you can put them into a *test fixture* class.

A *test program* can contain multiple test suites.

We'll now explain how to write a test program, starting at the individual assertion level and building up to tests and test suites.

### 7.4. Assertions

googletest assertions are macros that resemble function calls. You test a class or function by making assertions about its behavior. When an assertion fails, googletest prints the assertion's source file and line number location, along with a failure message. You may also supply a custom failure message which will be appended to googletest's message.

The assertions come in pairs that test the same thing but have different effects on the current function. **ASSERT\_\*** versions generate fatal failures when they fail, and **abort the current function**. **EXPECT\_\*** versions generate nonfatal failures, which don't abort the current function. Usually **EXPECT\_\***

are preferred, as they allow more than one failure to be reported in a test. However, you should use `ASSERT_*` if it doesn't make sense to continue when the assertion in question fails.

Since a failed `ASSERT_*` returns from the current function immediately, possibly skipping clean-up code that comes after it, it may cause a space leak. Depending on the nature of the leak, it may or may not be worth fixing - so keep this in mind if you get a heap checker error in addition to assertion errors.

To provide a custom failure message, simply stream it into the macro using the `<<` operator or a sequence of such operators. An example:

```
ASSERT_EQ(x.size(), y.size()) << "Vectors x and y are of unequal length";

for (int i = 0; i < x.size(); ++i) {
    EXPECT_EQ(x[i], y[i]) << "Vectors x and y differ at index " << i;
}
```

Anything that can be streamed to an `ostream` can be streamed to an assertion macro—in particular, C strings and `string` objects. If a wide string (`wchar_t*`, `TCHAR*` in UNICODE mode on Windows, or `std::wstring`) is streamed to an assertion, it will be translated to UTF-8 when printed.

#### 7.4.1. Basic Assertions

These assertions do basic true/false condition testing.

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_TRUE(condition);</code>	<code>EXPECT_TRUE(condition);</code>	<code>condition</code> is true
<code>ASSERT_FALSE(condition);</code>	<code>EXPECT_FALSE(condition);</code>	<code>condition</code> is false

Remember, when they fail, `ASSERT_*` yields a fatal failure and returns from the current function, while `EXPECT_*` yields a nonfatal failure, allowing the function to continue running. In either case, an assertion failure means its containing test fails.

**Availability:** Linux, Windows, Mac.

#### 7.4.2. Binary Comparison

This section describes assertions that compare two values.

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_EQ(val1, val2);</code>	<code>EXPECT_EQ(val1, val2);</code>	<code>val1 == val2</code>
<code>ASSERT_NE(val1, val2);</code>	<code>EXPECT_NE(val1, val2);</code>	<code>val1 != val2</code>
<code>ASSERT_LT(val1, val2);</code>	<code>EXPECT_LT(val1, val2);</code>	<code>val1 &lt; val2</code>
<code>ASSERT_LE(val1, val2);</code>	<code>EXPECT_LE(val1, val2);</code>	<code>val1 &lt;= val2</code>

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_GT(val1, val2);</code>	<code>EXPECT_GT(val1, val2);</code>	<code>val1 &gt; val2</code>
<code>ASSERT_GE(val1, val2);</code>	<code>EXPECT_GE(val1, val2);</code>	<code>val1 &gt;= val2</code>

Value arguments must be comparable by the assertion's comparison operator or you'll get a compiler error. We used to require the arguments to support the `<<` operator for streaming to an `ostream`, but this is no longer necessary. If `<<` is supported, it will be called to print the arguments when the assertion fails; otherwise googletest will attempt to print them in the best way it can. For more details and how to customize the printing of the arguments, see the documentation.

These assertions can work with a user-defined type, but only if you define the corresponding comparison operator (e.g., `==` or `<`). Since this is discouraged by the Google C++ Style Guide, you may need to use `ASSERT_TRUE()` or `EXPECT_TRUE()` to assert the equality of two objects of a user-defined type.

However, when possible, `ASSERT_EQ(actual, expected)` is preferred to `ASSERT_TRUE(actual == expected)`, since it tells you `actual` and `expected`'s values on failure.

Arguments are always evaluated exactly once. Therefore, it's OK for the arguments to have side effects. However, as with any ordinary C/C++ function, the arguments' evaluation order is undefined (i.e., the compiler is free to choose any order), and your code should not depend on any particular argument evaluation order.

`ASSERT_EQ()` does pointer equality on pointers. If used on two C strings, it tests if they are in the same memory location, not if they have the same value. Therefore, if you want to compare C strings (e.g. `const char*`) by value, use `ASSERT_STREQ()`, which will be described later on. In particular, to assert that a C string is NULL, use `ASSERT_STREQ(c_string, NULL)`. Consider using `ASSERT_EQ(c_string, nullptr)` if c++11 is supported. To compare two `string` objects, you should use `ASSERT_EQ`.

When doing pointer comparisons use `*_EQ(ptr, nullptr)` and `*_NE(ptr, nullptr)` instead of `*_EQ(ptr, NULL)` and `*_NE(ptr, NULL)`. This is because `nullptr` is typed, while `NULL` is not. See the FAQ for more details.

If you're working with floating point numbers, you may want to use the floating point variations of some of these macros in order to avoid problems caused by rounding. See Advanced googletest Topics for details.

Macros in this section work with both narrow and wide string objects (`string` and `wstring`).

**Availability:** Linux, Windows, Mac.

**Historical note:** Before February 2016 `*_EQ` had a convention of calling it as `ASSERT_EQ(expected, actual)`, so lots of existing code uses this order. Now `*_EQ` treats both parameters in the same way.



### 7.4.3. String Comparison

The assertions in this group compare two **C strings**. If you want to compare two **string** objects, use `EXPECT_EQ`, `EXPECT_NE`, and etc instead.

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_STREQ(str1, str2);</code>	<code>EXPECT_STREQ(str1, str2);</code>	the two C strings have the same content
<code>ASSERT_STRNE(str1, str2);</code>	<code>EXPECT_STRNE(str1, str2);</code>	the two C strings have different contents
<code>ASSERT_STRCASEEQ(str1, str2);</code>	<code>EXPECT_STRCASEEQ(str1, str2);</code>	the two C strings have the same content, ignoring case
<code>ASSERT_STRCASENE(str1, str2);</code>	<code>EXPECT_STRCASENE(str1, str2);</code>	the two C strings have different contents, ignoring case

Note that “CASE” in an assertion name means that case is ignored. A `NULL` pointer and an empty string are considered *different*.

`*STREQ*` and `*STRNE*` also accept wide C strings (`wchar_t*`). If a comparison of two wide strings fails, their values will be printed as UTF-8 narrow strings.

**Availability:** Linux, Windows, Mac.

**See also:** For more string comparison tricks (substring, prefix, suffix, and regular expression matching, for example), see this in the Advanced googletest Guide.

## 7.5. Simple Tests

To create a test:

1. Use the `TEST()` macro to define and name a test function. These are ordinary C++ functions that don’t return a value.
2. In this function, along with any valid C++ statements you want to include, use the various googletest assertions to check values.
3. The test’s result is determined by the assertions; if any assertion in the test fails (either fatally or non-fatally), or if the test crashes, the entire test fails. Otherwise, it succeeds.

```
TEST(TestSuiteName, TestName) {  
    ... test body ...  
}
```

`TEST()` arguments go from general to specific. The *first* argument is the name of the test suite, and the *second* argument is the test’s name within the test suite. Both names must be valid C++ identifiers, and they should not contain any underscores (`_`). A test’s *full name* consists of its containing test suite and its individual name. Tests from different test suites can have the same individual name.

For example, let’s take a simple integer function:

```
int Factorial(int n);    // Returns the factorial of n
```

A test suite for this function might look like:

```
// Tests factorial of 0.  
TEST(FactorialTest, HandlesZeroInput) {
```

```

    EXPECT_EQ(Factorial(0), 1);
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
    EXPECT_EQ(Factorial(1), 1);
    EXPECT_EQ(Factorial(2), 2);
    EXPECT_EQ(Factorial(3), 6);
    EXPECT_EQ(Factorial(8), 40320);
}

```

googletest groups the test results by test suites, so logically related tests should be in the same test suite; in other words, the first argument to their `TEST()` should be the same. In the above example, we have two tests, `HandlesZeroInput` and `HandlesPositiveInput`, that belong to the same test suite `FactorialTest`.

When naming your test suites and tests, you should follow the same convention as for naming functions and classes.

**Availability:** Linux, Windows, Mac.

## 7.6. Test Fixtures: Using the Same Data Configuration for Multiple Tests

If you find yourself writing two or more tests that operate on similar data, you can use a *test fixture*. This allows you to reuse the same configuration of objects for several different tests.

To create a fixture:

1. Derive a class from `::testing::Test`. Start its body with `protected:`, as we'll want to access fixture members from sub-classes.
2. Inside the class, declare any objects you plan to use.
3. If necessary, write a default constructor or `SetUp()` function to prepare the objects for each test. A common mistake is to spell `SetUp()` as `Setup()` with a small u - Use `override` in C++11 to make sure you spelled it correctly.
4. If necessary, write a destructor or `TearDown()` function to release any resources you allocated in `SetUp()`. To learn when you should use the constructor/destructor and when you should use `SetUp()/TearDown()`, read the FAQ.
5. If needed, define subroutines for your tests to share.

When using a fixture, use `TEST_F()` instead of `TEST()` as it allows you to access objects and subroutines in the test fixture:

```

TEST_F(TestFixtureName, TestName) {
    ... test body ...
}

```

Like `TEST()`, the first argument is the test suite name, but for `TEST_F()` this must be the name of the test fixture class. You've probably guessed: `_F` is for

fixture.

Unfortunately, the C++ macro system does not allow us to create a single macro that can handle both types of tests. Using the wrong macro causes a compiler error.

Also, you must first define a test fixture class before using it in a `TEST_F()`, or you'll get the compiler error “`virtual outside class declaration`”.

For each test defined with `TEST_F()`, googletest will create a *fresh* test fixture at runtime, immediately initialize it via `SetUp()`, run the test, clean up by calling `TearDown()`, and then delete the test fixture. Note that different tests in the same test suite have different test fixture objects, and googletest always deletes a test fixture before it creates the next one. googletest does **not** reuse the same test fixture for multiple tests. Any changes one test makes to the fixture do not affect other tests.

As an example, let's write tests for a FIFO queue class named `Queue`, which has the following interface:

```
template <typename E> // E is the element type.
class Queue {
public:
    Queue();
    void Enqueue(const E& element);
    E* Dequeue(); // Returns NULL if the queue is empty.
    size_t size() const;
    ...
};
```

First, define a fixture class. By convention, you should give it the name `FooTest` where `Foo` is the class being tested.

```
class QueueTest : public ::testing::Test {
protected:
    void SetUp() override {
        q1_.Enqueue(1);
        q2_.Enqueue(2);
        q2_.Enqueue(3);
    }

    // void TearDown() override {}

    Queue<int> q0_;
    Queue<int> q1_;
    Queue<int> q2_;
};
```

In this case, `TearDown()` is not needed since we don't have to clean up after each test, other than what's already done by the destructor.

Now we'll write tests using `TEST_F()` and this fixture.

```
TEST_F(QueueTest, IsEmptyInitially) {
    EXPECT_EQ(q0_.size(), 0);
```

```

}

TEST_F(QueueTest, DequeueWorks) {
    int* n = q0_.Dequeue();
    EXPECT_EQ(n, nullptr);

    n = q1_.Dequeue();
    ASSERT_NE(n, nullptr);
    EXPECT_EQ(*n, 1);
    EXPECT_EQ(q1_.size(), 0);
    delete n;

    n = q2_.Dequeue();
    ASSERT_NE(n, nullptr);
    EXPECT_EQ(*n, 2);
    EXPECT_EQ(q2_.size(), 1);
    delete n;
}

```

The above uses both `ASSERT_*` and `EXPECT_*` assertions. The rule of thumb is to use `EXPECT_*` when you want the test to continue to reveal more errors after the assertion failure, and use `ASSERT_*` when continuing after failure doesn't make sense. For example, the second assertion in the `Dequeue` test is `ASSERT_NE(nullptr, n)`, as we need to dereference the pointer `n` later, which would lead to a segfault when `n` is `NULL`.

When these tests run, the following happens:

1. googletest constructs a `QueueTest` object (let's call it `t1`).
2. `t1.Setup()` initializes `t1`.
3. The first test (`IsEmptyInitially`) runs on `t1`.
4. `t1.TearDown()` cleans up after the test finishes.
5. `t1` is destructed.
6. The above steps are repeated on another `QueueTest` object, this time running the `DequeueWorks` test.

**Availability:** Linux, Windows, Mac.

## 7.7. Invoking the Tests

`TEST()` and `TEST_F()` implicitly register their tests with googletest. So, unlike with many other C++ testing frameworks, you don't have to re-list all your defined tests in order to run them.

After defining your tests, you can run them with `RUN_ALL_TESTS()`, which returns 0 if all the tests are successful, or 1 otherwise. Note that `RUN_ALL_TESTS()` runs *all tests* in your link unit—they can be from different test suites, or even different source files.

When invoked, the `RUN_ALL_TESTS()` macro:

- Saves the state of all googletest flags.
- Creates a test fixture object for the first test.

- Initializes it via `SetUp()`.
- Runs the test on the fixture object.
- Cleans up the fixture via `TearDown()`.
- Deletes the fixture.
- Restores the state of all googletest flags.
- Repeats the above steps for the next test, until all tests have run.

If a fatal failure happens the subsequent steps will be skipped.

IMPORTANT: You must **not** ignore the return value of `RUN_ALL_TESTS()`, or you will get a compiler error. The rationale for this design is that the automated testing service determines whether a test has passed based on its exit code, not on its stdout/stderr output; thus your `main()` function must return the value of `RUN_ALL_TESTS()`.

Also, you should call `RUN_ALL_TESTS()` only **once**. Calling it more than once conflicts with some advanced googletest features (e.g., thread-safe death tests) and thus is not supported.

**Availability:** Linux, Windows, Mac.

## 7.8. Writing the `main()` Function

Most users should *not* need to write their own `main` function and instead link with `gtest_main` (as opposed to with `gtest`), which defines a suitable entry point. See the end of this section for details. The remainder of this section should only apply when you need to do something custom before the tests run that cannot be expressed within the framework of fixtures and test suites.

If you write your own `main` function, it should return the value of `RUN_ALL_TESTS()`.

You can start from this boilerplate:

```
#include "this/package/foo.h"

#include "gtest/gtest.h"

namespace my {
namespace project {
namespace {

// The fixture for testing class Foo.
class FooTest : public ::testing::Test {
protected:
// You can remove any or all of the following functions if their bodies would
// be empty.

FooTest() {
// You can do set-up work for each test here.
```





```
}

~FooTest() override {
    // You can do clean-up work that doesn't throw exceptions here.
}

// If the constructor and destructor are not enough for setting up
// and cleaning up each test, you can define the following methods:

void SetUp() override {
    // Code here will be called immediately after the constructor (right
    // before each test).
}

void TearDown() override {
    // Code here will be called immediately after each test (right
    // before the destructor).
}

// Class members declared here can be used by all tests in the test suite
// for Foo.
};

// Tests that the Foo::Bar() method does Abc.
TEST_F(FooTest, MethodBarDoesAbc) {
    const std::string input_filepath = "this/package/testdata/myinputfile.dat";
    const std::string output_filepath = "this/package/testdata/myoutputfile.dat";
    Foo f;
    EXPECT_EQ(f.Bar(input_filepath, output_filepath), 0);
}

// Tests that Foo does Xyz.
TEST_F(FooTest, DoesXyz) {
    // Exercises the Xyz feature of Foo.
}

} // namespace
} // namespace project
} // namespace my

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

The `::testing::InitGoogleTest()` function parses the command line for googletest flags, and removes all recognized flags. This allows the user to control a test program's behavior via various flags, which we'll cover in the AdvancedGuide. You **must** call this function before calling `RUN_ALL_TESTS()`, or the flags won't be properly initialized.

On Windows, `InitGoogleTest()` also works with wide strings, so it can be



used in programs compiled in UNICODE mode as well.

But maybe you think that writing all those `main` functions is too much work? We agree with you completely, and that's why Google Test provides a basic implementation of `main()`. If it fits your needs, then just link your test with the `gtest_main` library and you are good to go.

NOTE: `ParseGUnitFlags()` is deprecated in favor of `InitGoogleTest()`.

## 7.9. Known Limitations

- Google Test is designed to be thread-safe. The implementation is thread-safe on systems where the `pthread` library is available. It is currently *unsafe* to use Google Test assertions from two threads concurrently on other systems (e.g. Windows). In most tests this is not an issue as usually the assertions are done in the main thread. If you want to help, you can volunteer to implement the necessary synchronization primitives in `gtest-port.h` for your platform.

## 8. Advanced googletest Topics

### 8.1. Introduction

Now that you have read the googletest Primer and learned how to write tests using googletest, it's time to learn some new tricks. This document will show you more assertions as well as how to construct complex failure messages, propagate fatal failures, reuse and speed up your test fixtures, and use various flags with your tests.

### 8.2. More Assertions

This section covers some less frequently used, but still significant, assertions.

#### 8.2.1. Explicit Success and Failure

These three assertions do not actually test a value or expression. Instead, they generate a success or failure directly. Like the macros that actually perform a test, you may stream a custom failure message into them.

`SUCCEED();`

Generates a success. This does **NOT** make the overall test succeed. A test is considered successful only if none of its assertions fail during its execution.

NOTE: `SUCCEED()` is purely documentary and currently doesn't generate any user-visible output. However, we may add `SUCCEED()` messages to googletest's output in the future.

`FAIL();`

`ADD_FAILURE();`

`ADD_FAILURE_AT("file_path", line_number);`

`FAIL()` generates a fatal failure, while `ADD_FAILURE()` and `ADD_FAILURE_AT()` generate a nonfatal failure. These are useful when control flow, rather than a Boolean expression, determines the test's success or failure. For example, you might want to write something like:

```
switch(expression) {
  case 1:
    ... some checks ...
  case 2:
    ... some other checks ...
  default:
    FAIL() << "We shouldn't get here.";
}
```

NOTE: you can only use `FAIL()` in functions that return `void`. See the Assertion Placement section for more information.

#### 8.2.2. Exception Assertions

These are for verifying that a piece of code throws (or does not throw) an exception of the given type:



Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_THROW(statement, exception_type);</code>	<code>EXPECT_THROW(statement, exception_type);</code>	statement throws an exception of the given type
<code>ASSERT_ANY_THROW(statement);</code>	<code>EXPECT_ANY_THROW(statement);</code>	statement throws an exception of any type
<code>ASSERT_NO_THROW(statement);</code>	<code>EXPECT_NO_THROW(statement);</code>	statement doesn't throw any exception

Examples:

```
ASSERT_THROW(Foo(5), bar_exception);
```

```
EXPECT_NO_THROW({  
    int n = 5;  
    Bar(&n);  
});
```

**Availability:** requires exceptions to be enabled in the build environment

### 8.2.3. Predicate Assertions for Better Error Messages

Even though googletest has a rich set of assertions, they can never be complete, as it's impossible (nor a good idea) to anticipate all scenarios a user might run into. Therefore, sometimes a user has to use `EXPECT_TRUE()` to check a complex expression, for lack of a better macro. This has the problem of not showing you the values of the parts of the expression, making it hard to understand what went wrong. As a workaround, some users choose to construct the failure message by themselves, streaming it into `EXPECT_TRUE()`. However, this is awkward especially when the expression has side-effects or is expensive to evaluate.

googletest gives you three different options to solve this problem:

**Using an Existing Boolean Function** If you already have a function or functor that returns `bool` (or a type that can be implicitly converted to `bool`), you can use it in a *predicate assertion* to get the function arguments printed for free:

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_PRED1(pred1, val1)</code>	<code>EXPECT_PRED1(pred1, val1)</code>	<code>pred1(val1)</code> is true
<code>ASSERT_PRED2(pred2, val1, val2)</code>	<code>EXPECT_PRED2(pred2, val1, val2)</code>	<code>pred2(val1, val2)</code> is true
...	...	...

In the above, `predn` is an `n`-ary predicate function or functor, where `val1`, `val2`, ..., and `valn` are its arguments. The assertion succeeds if the predicate returns `true` when applied to the given arguments, and fails otherwise. When the assertion fails, it prints the value of each argument. In either case, the arguments are evaluated exactly once.

Here's an example. Given

```
// Returns true if m and n have no common divisors except 1.
bool MutuallyPrime(int m, int n) { ... }

const int a = 3;
const int b = 4;
const int c = 10;
```

the assertion

```
EXPECT_PRED2(MutuallyPrime, a, b);
```

will succeed, while the assertion

```
EXPECT_PRED2(MutuallyPrime, b, c);
```

will fail with the message

```
MutuallyPrime(b, c) is false, where
b is 4
c is 10
```

NOTE:

1. If you see a compiler error “no matching function to call” when using `ASSERT_PRED*` or `EXPECT_PRED*`, please see this for how to resolve it.

**Using a Function That Returns an `AssertionResult`** While `EXPECT_PRED*()` and friends are handy for a quick job, the syntax is not satisfactory: you have to use different macros for different arities, and it feels more like Lisp than C++. The `::testing::AssertionResult` class solves this problem.

An `AssertionResult` object represents the result of an assertion (whether it's a success or a failure, and an associated message). You can create an `AssertionResult` using one of these factory functions:

```
namespace testing {  
  
    // Returns an AssertionResult object to indicate that an assertion has  
    // succeeded.  
    AssertionResult AssertionSuccess();  
  
    // Returns an AssertionResult object to indicate that an assertion has  
    // failed.  
    AssertionResult AssertionFailure();  
  
}
```

You can then use the `<<` operator to stream messages to the `AssertionResult` object.

To provide more readable messages in Boolean assertions (e.g. `EXPECT_TRUE()`), write a predicate function that returns `AssertionResult` instead of `bool`. For example, if you define `IsEven()` as:

```
testing::AssertionResult IsEven(int n) {  
    if ((n % 2) == 0)  
        return testing::AssertionSuccess();  
    else  
        return testing::AssertionFailure() << n << " is odd";  
}
```

instead of:

```
bool IsEven(int n) {  
    return (n % 2) == 0;  
}
```

the failed assertion `EXPECT_TRUE(IsEven(Fib(4)))` will print:

```
Value of: IsEven(Fib(4))  
  Actual: false (3 is odd)  
Expected: true
```

instead of a more opaque

```
Value of: IsEven(Fib(4))  
  Actual: false  
Expected: true
```

If you want informative messages in `EXPECT_FALSE` and `ASSERT_FALSE` as well (one third of Boolean assertions in the Google code base are negative ones), and are fine with making the predicate slower in the success case, you can supply a success message:

```
testing::AssertionResult IsEven(int n) {  
    if ((n % 2) == 0)  
        return testing::AssertionSuccess() << n << " is even";  
}
```

```

else
    return testing::AssertionFailure() << n << " is odd";
}

```

Then the statement `EXPECT_FALSE(IsEven(Fib(6)))` will print

```

Value of: IsEven(Fib(6))
Actual: true (8 is even)
Expected: false

```

**Using a Predicate-Formatter** If you find the default message generated by `(ASSERT|EXPECT)_PRED*` and `(ASSERT|EXPECT)_(TRUE|FALSE)` unsatisfactory, or some arguments to your predicate do not support streaming to `ostream`, you can instead use the following *predicate-formatter* assertions to fully customize how the message is formatted:

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_PRED_FORMAT1(pred_format1, val1);</code>	<code>EXPECT_PRED_FORMAT1(pred_format1, val1);</code>	<code>pred_format1(val1)</code> is successful
<code>ASSERT_PRED_FORMAT2(pred_format2, val1, val2);</code>	<code>EXPECT_PRED_FORMAT2(pred_format2, val1, val2);</code>	<code>pred_format2(val1, val2)</code> is successful
...	...	...

The difference between this and the previous group of macros is that instead of a predicate, `(ASSERT|EXPECT)_PRED_FORMAT*` take a *predicate-formatter* (`pred_formatn`), which is a function or functor with the signature:

```

testing::AssertionResult PredicateFormatter(const char* expr1,
                                           const char* expr2,
                                           ...
                                           const char* exprn,
                                           T1 val1,
                                           T2 val2,
                                           ...
                                           Tn valn);

```

where `val1`, `val2`, ..., and `valn` are the values of the predicate arguments, and `expr1`, `expr2`, ..., and `exprn` are the corresponding expressions as they appear in the source code. The types `T1`, `T2`, ..., and `Tn` can be either value types or reference types. For example, if an argument has type `Foo`, you can declare it as either `Foo` or `const Foo&`, whichever is appropriate.

As an example, let's improve the failure message in `MutuallyPrime()`, which was used with `EXPECT_PRED2()`:



```
// Returns the smallest prime common divisor of m and n,
// or 1 when m and n are mutually prime.
int SmallestPrimeCommonDivisor(int m, int n) { ... }

// A predicate-formatter for asserting that two integers are mutually prime.
testing::AssertionResult AssertMutuallyPrime(const char* m_expr,
                                             const char* n_expr,
                                             int m,
                                             int n) {
    if (MutuallyPrime(m, n)) return testing::AssertionSuccess();

    return testing::AssertionFailure() << m_expr << " and " << n_expr
        << " (" << m << " and " << n << ") are not mutually prime, "
        << "as they have a common divisor " << SmallestPrimeCommonDivisor(m, n);
}
```

With this predicate-formatter, we can use

```
EXPECT_PRED_FORMAT2(AssertMutuallyPrime, b, c);
```

to generate the message

b and c (4 and 10) are not mutually prime, as they have a common divisor 2.

As you may have realized, many of the built-in assertions we introduced earlier are special cases of (EXPECT|ASSERT)\_PRED\_FORMAT\*. In fact, most of them are indeed defined using (EXPECT|ASSERT)\_PRED\_FORMAT\*.

#### 8.2.4. Floating-Point Comparison

Comparing floating-point numbers is tricky. Due to round-off errors, it is very unlikely that two floating-points will match exactly. Therefore, ASSERT\_EQ's naive comparison usually doesn't work. And since floating-points can have a wide value range, no single fixed error bound works. It's better to compare by a fixed relative error bound, except for values close to 0 due to the loss of precision there.

In general, for floating-point comparison to make sense, the user needs to carefully choose the error bound. If they don't want or care to, comparing in terms of Units in the Last Place (ULPs) is a good default, and googletest provides assertions to do this. Full details about ULPs are quite long; if you want to learn more, see [here](#).

#### Floating-Point Macros

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_FLOAT_EQ(val1, val2);	EXPECT_FLOAT_EQ(val1, val2);	the two float values are almost equal
ASSERT_DOUBLE_EQ(val1, val2);	EXPECT_DOUBLE_EQ(val1, val2);	the two double values are almost equal





By “almost equal” we mean the values are within 4 ULP’s from each other. The following assertions allow you to choose the acceptable error bound:

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_NEAR(val1, val2, abs_error);</code>	<code>EXPECT_NEAR(val1, val2, abs_error);</code>	the difference between <code>val1</code> and <code>val2</code> doesn’t exceed the given absolute error

**Floating-Point Predicate-Format Functions** Some floating-point operations are useful, but not that often used. In order to avoid an explosion of new macros, we provide them as predicate-format functions that can be used in predicate assertion macros (e.g. `EXPECT_PRED_FORMAT2`, etc).

```
EXPECT_PRED_FORMAT2(testing::FloatLE, val1, val2);
EXPECT_PRED_FORMAT2(testing::DoubleLE, val1, val2);
```

Verifies that `val1` is less than, or almost equal to, `val2`. You can replace `EXPECT_PRED_FORMAT2` in the above table with `ASSERT_PRED_FORMAT2`.

### 8.2.5. Asserting Using gMock Matchers

gMock comes with a library of matchers for validating arguments passed to mock objects. A gMock *matcher* is basically a predicate that knows how to describe itself. It can be used in these assertion macros:

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_THAT(value, matcher);</code>	<code>EXPECT_THAT(value, matcher);</code>	value matches matcher

For example, `StartsWith(prefix)` is a matcher that matches a string starting with `prefix`, and you can write:

```
using ::testing::StartsWith;
...
// Verifies that Foo() returns a string starting with "Hello".
EXPECT_THAT(Foo(), StartsWith("Hello"));
```

Read this recipe in the gMock Cookbook for more details.

gMock has a rich set of matchers. You can do many things googletest cannot do alone with them. For a list of matchers gMock provides, read this. It’s easy to write your own matchers too.

gMock is bundled with googletest, so you don’t need to add any build dependency in order to take advantage of this. Just include `"gmock/gmock.h"` and you’re ready to go.



### 8.2.6. More String Assertions

(Please read the previous section first if you haven't.)

You can use the gMock string matchers with `EXPECT_THAT()` or `ASSERT_THAT()` to do more string comparison tricks (sub-string, prefix, suffix, regular expression, and etc). For example,

```
using ::testing::HasSubstr;
using ::testing::MatchesRegex;
...
ASSERT_THAT(foo_string, HasSubstr("needle"));
EXPECT_THAT(bar_string, MatchesRegex("\\w*\\d+"));
```

If the string contains a well-formed HTML or XML document, you can check whether its DOM tree matches an XPath expression:

```
// Currently still in //template/prototemplate/testing/xpath_matcher
#include "template/prototemplate/testing/xpath_matcher.h"
using ::prototemplate::testing::MatchesXPath;
EXPECT_THAT(html_string, MatchesXPath("//a[text()='click here']"));
```

### 8.2.7. Windows HRESULT assertions

These assertions test for HRESULT success or failure.

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_HRESULT_SUCCEEDED(expression)</code>	<code>EXPECT_HRESULT_SUCCEEDED(expression)</code>	<code>expression</code> is a success HRESULT
<code>ASSERT_HRESULT_FAILED(expression)</code>	<code>EXPECT_HRESULT_FAILED(expression)</code>	<code>expression</code> is a failure HRESULT

The generated output contains the human-readable error message associated with the HRESULT code returned by `expression`.

You might use them like this:

```
COMPtr<IShellDispatch2> shell;
ASSERT_HRESULT_SUCCEEDED(shell.CoCreateInstance(L"Shell.Application"));
COMVariant empty;
ASSERT_HRESULT_SUCCEEDED(shell->ShellExecute(CComBSTR(url), empty, empty, empty, empty
```

### 8.2.8. Type Assertions

You can call the function

```
::testing::StaticAssertTypeEq<T1, T2>();
```

to assert that types `T1` and `T2` are the same. The function does nothing if the assertion is satisfied. If the types are different, the function call will fail to compile, the compiler error message will say that `T1` and `T2` are not the same type and most likely (depending on the compiler) show you the actual values of `T1` and `T2`. This is mainly useful inside template code.

**Caveat:** When used inside a member function of a class template or a function template, `StaticAssertTypeEq<T1, T2>()` is effective only if the function is instantiated. For example, given:

```
template <typename T> class Foo {  
    public:  
        void Bar() { testing::StaticAssertTypeEq<int, T>(); }  
};
```

the code:

```
void Test1() { Foo<bool> foo; }
```

will not generate a compiler error, as `Foo<bool>::Bar()` is never actually instantiated. Instead, you need:

```
void Test2() { Foo<bool> foo; foo.Bar(); }
```

to cause a compiler error.

### 8.2.9. Assertion Placement

You can use assertions in any C++ function. In particular, it doesn't have to be a method of the test fixture class. The one constraint is that assertions that generate a fatal failure (`FAIL*` and `ASSERT_*`) can only be used in void-returning functions. This is a consequence of Google's not using exceptions. By placing it in a non-void function you'll get a confusing compile error like `.error: void value not ignored as it ought to be` or `cannot initialize return object of type 'bool' with an rvalue of type 'void'` or `.error: no viable conversion from 'void' to 'string'`.

If you need to use fatal assertions in a function that returns non-void, one option is to make the function return the value in an out parameter instead. For example, you can rewrite `T2 Foo(T1 x)` to `void Foo(T1 x, T2* result)`. You need to make sure that `*result` contains some sensible value even when the function returns prematurely. As the function now returns `void`, you can use any assertion inside of it.

If changing the function's type is not an option, you should just use assertions that generate non-fatal failures, such as `ADD_FAILURE*` and `EXPECT_*`.

**NOTE:** Constructors and destructors are not considered void-returning functions, according to the C++ language specification, and so you may not use fatal assertions in them; you'll get a compilation error if you try. Instead, either call `abort` and crash the entire test executable, or put the fatal assertion in a `SetUp/TearDown` function; see `constructor/destructor vs. ~SetUp/TearDown`

**WARNING:** A fatal assertion in a helper function (private void-returning method) called from a constructor or destructor does not terminate the current test, as your intuition might suggest: it merely returns from the constructor or destructor early, possibly leaving your object in a partially-constructed or partially-destructed state! You almost certainly want to `abort` or use `SetUp/TearDown` instead.

### 8.3. Teaching googletest How to Print Your Values

When a test assertion such as `EXPECT_EQ` fails, googletest prints the argument values to help you debug. It does this using a user-extensible value printer.

This printer knows how to print built-in C++ types, native arrays, STL containers, and any type that supports the `<<` operator. For other types, it prints the raw bytes in the value and hopes that you the user can figure it out.

As mentioned earlier, the printer is *extensible*. That means you can teach it to do a better job at printing your particular type than to dump the bytes. To do that, define `<<` for your type:

```
#include <ostream>

namespace foo {

class Bar { // We want googletest to be able to print instances of this.
...
    // Create a free inline friend function.
    friend std::ostream& operator<<(std::ostream& os, const Bar& bar) {
        return os << bar.DebugString(); // whatever needed to print bar to os
    }
};

// If you can't declare the function in the class it's important that the
// << operator is defined in the SAME namespace that defines Bar. C++'s look-up
// rules rely on that.
std::ostream& operator<<(std::ostream& os, const Bar& bar) {
    return os << bar.DebugString(); // whatever needed to print bar to os
}

} // namespace foo
```

Sometimes, this might not be an option: your team may consider it bad style to have a `<<` operator for `Bar`, or `Bar` may already have a `<<` operator that doesn't do what you want (and you cannot change it). If so, you can instead define a `PrintTo()` function like this:

```
#include <ostream>

namespace foo {

class Bar {
...
    friend void PrintTo(const Bar& bar, std::ostream* os) {
        *os << bar.DebugString(); // whatever needed to print bar to os
    }
};

// If you can't declare the function in the class it's important that PrintTo()
// is defined in the SAME namespace that defines Bar. C++'s look-up rules rely
// on that.
```



```
void PrintTo(const Bar& bar, std::ostream* os) {  
    *os << bar.DebugString(); // whatever needed to print bar to os  
}  
  
} // namespace foo
```

If you have defined both `<<` and `PrintTo()`, the latter will be used when googletest is concerned. This allows you to customize how the value appears in googletest's output without affecting code that relies on the behavior of its `<<` operator.

If you want to print a value `x` using googletest's value printer yourself, just call `::testing::PrintToString(x)`, which returns an `std::string`:

```
vector<pair<Bar, int> > bar_ints = GetBarIntVector();  
  
EXPECT_TRUE(IsCorrectBarIntVector(bar_ints))  
    << "bar_ints = " << testing::PrintToString(bar_ints);
```

## 8.4. Death Tests

In many applications, there are assertions that can cause application failure if a condition is not met. These sanity checks, which ensure that the program is in a known good state, are there to fail at the earliest possible time after some program state is corrupted. If the assertion checks the wrong condition, then the program may proceed in an erroneous state, which could lead to memory corruption, security holes, or worse. Hence it is vitally important to test that such assertion statements work as expected.

Since these precondition checks cause the processes to die, we call such tests *death tests*. More generally, any test that checks that a program terminates (except by throwing an exception) in an expected fashion is also a death test.

Note that if a piece of code throws an exception, we don't consider it "death" for the purpose of death tests, as the caller of the code could catch the exception and avoid the crash. If you want to verify exceptions thrown by your code, see [Exception Assertions](#).

If you want to test `EXPECT_*()/ASSERT_*` failures in your test code, see [Catching Failures](#)

### 8.4.1. How to Write a Death Test

googletest has the following macros to support death tests:

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_DEATH(statement, matcher);</code>	<code>EXPECT_DEATH(statement, matcher);</code>	statement crashes with the given error



Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_DEATH_IF_SUPPORTED(statement, matcher);</code>	<code>EXPECT_DEATH_IF_SUPPORTED(statement, matcher);</code>	if death tests are supported, verifies that <b>statement</b> crashes with the given error; otherwise verifies nothing
<code>ASSERT_DEBUG_DEATH(statement, matcher);</code>	<code>EXPECT_DEBUG_DEATH(statement, matcher);</code>	<b>statement</b> crashes with the given error in <b>debug mode</b> . When not in debug (i.e. <code>NDEBUG</code> is defined), this just executes <b>statement</b>
<code>ASSERT_EXIT(statement, predicate, matcher);</code>	<code>EXPECT_EXIT(statement, predicate, matcher);</code>	<b>statement</b> exits with the given error, and its exit code matches <b>predicate</b>

where **statement** is a statement that is expected to cause the process to die, **predicate** is a function or function object that evaluates an integer exit status, and **matcher** is either a gMock matcher matching a `const std::string&` or a (Perl) regular expression - either of which is matched against the stderr output of **statement**. For legacy reasons, a bare string (i.e. with no matcher) is interpreted as `ContainsRegex(str)`, **not** `Eq(str)`. Note that **statement** can be *any valid statement* (including *compound statement*) and doesn't have to be an expression.

As usual, the `ASSERT` variants abort the current test function, while the `EXPECT` variants do not.

NOTE: We use the word “crash” here to mean that the process terminates with a *non-zero* exit status code. There are two possibilities: either the process has called `exit()` or `_exit()` with a non-zero value, or it may be killed by a signal.

This means that if **statement** terminates the process with a 0 exit code, it is *not* considered a crash by `EXPECT_DEATH`. Use `EXPECT_EXIT` instead if this is the case, or if you want to restrict the exit code more precisely.

A predicate here must accept an `int` and return a `bool`. The death test succeeds only if the predicate returns `true`. googletest defines a few predicates that handle the most common cases:

```
::testing::ExitedWithCode(exit_code)
```

This expression is `true` if the program exited normally with the given exit code.

```
testing::KilledBySignal(signal_number) // Not available on Windows.
```

This expression is `true` if the program was killed by the given signal.

The `*_DEATH` macros are convenient wrappers for `*_EXIT` that use a predicate that verifies the process' exit code is non-zero.

Note that a death test only cares about three things:

1. does `statement` abort or exit the process?
2. (in the case of `ASSERT_EXIT` and `EXPECT_EXIT`) does the exit status satisfy `predicate`? Or (in the case of `ASSERT_DEATH` and `EXPECT_DEATH`) is the exit status non-zero? And
3. does the stderr output match `matcher`?

In particular, if `statement` generates an `ASSERT_*` or `EXPECT_*` failure, it will **not** cause the death test to fail, as googletest assertions don't abort the process.

To write a death test, simply use one of the above macros inside your test function. For example,

```
TEST(MyDeathTest, Foo) {  
    // This death test uses a compound statement.  
    ASSERT_DEATH({  
        int n = 5;  
        Foo(&n);  
    }, "Error on line .* of Foo()");  
}  
  
TEST(MyDeathTest, NormalExit) {  
    EXPECT_EXIT(NormalExit(), testing::ExitedWithCode(0), "Success");  
}  
  
TEST(MyDeathTest, KillMyself) {  
    EXPECT_EXIT(KillMyself(), testing::KilledBySignal(SIGKILL),  
                "Sending myself unblockable signal");  
}
```

verifies that:

- calling `Foo(5)` causes the process to die with the given error message,
- calling `NormalExit()` causes the process to print "Success" to stderr and exit with exit code 0, and
- calling `KillMyself()` kills the process with signal `SIGKILL`.

The test function body may contain other assertions and statements as well, if necessary.

#### 8.4.2. Death Test Naming

IMPORTANT: We strongly recommend you to follow the convention of naming your **test suite** (not test) `*DeathTest` when it contains a death test,

as demonstrated in the above example. The Death Tests And Threads section below explains why.

If a test fixture class is shared by normal tests and death tests, you can use `using` or `typedef` to introduce an alias for the fixture class and avoid duplicating its code:

```
class FooTest : public testing::Test { ... };

using FooDeathTest = FooTest;

TEST_F(FooTest, DoesThis) {
    // normal test
}

TEST_F(FooDeathTest, DoesThat) {
    // death test
}
```

### 8.4.3. Regular Expression Syntax

On POSIX systems (e.g. Linux, Cygwin, and Mac), googletest uses the POSIX extended regular expression syntax. To learn about this syntax, you may want to read this [Wikipedia entry](#).

On Windows, googletest uses its own simple regular expression implementation. It lacks many features. For example, we don't support union ("`x|y`"), grouping ("`(xy)`"), brackets ("`[xy]`"), and repetition count ("`x{5,7}`"), among others. Below is what we do support (A denotes a literal character, period (`.`), or a single `\\` escape sequence; `x` and `y` denote regular expressions.):

Expression	Meaning
<code>c</code>	matches any literal character <code>c</code>
<code>\\d</code>	matches any decimal digit
<code>\\D</code>	matches any character that's not a decimal digit
<code>\\f</code>	matches <code>\\f</code>
<code>\\n</code>	matches <code>\\n</code>
<code>\\r</code>	matches <code>\\r</code>
<code>\\s</code>	matches any ASCII whitespace, including <code>\\n</code>
<code>\\S</code>	matches any character that's not a whitespace
<code>\\t</code>	matches <code>\\t</code>
<code>\\v</code>	matches <code>\\v</code>
<code>\\w</code>	matches any letter, <code>_</code> , or decimal digit
<code>\\W</code>	matches any character that <code>\\w</code> doesn't match
<code>\\c</code>	matches any literal character <code>c</code> , which must be a punctuation
<code>.</code>	matches any single character except <code>\\n</code>



Expression	Meaning
<code>A?</code>	matches 0 or 1 occurrences of <code>A</code>
<code>A*</code>	matches 0 or many occurrences of <code>A</code>
<code>A+</code>	matches 1 or many occurrences of <code>A</code>
<code>^</code>	matches the beginning of a string (not that of each line)
<code>\$</code>	matches the end of a string (not that of each line)
<code>xy</code>	matches <code>x</code> followed by <code>y</code>

To help you determine which capability is available on your system, googletest defines macros to govern which regular expression it is using. The macros are: `GTEST_USES_SIMPLE_RE=1` or `GTEST_USES_POSIX_RE=1`. If you want your death tests to work in all cases, you can either `#if` on these macros or use the more limited syntax only.

#### 8.4.4. How It Works

Under the hood, `ASSERT_EXIT()` spawns a new process and executes the death test statement in that process. The details of how precisely that happens depend on the platform and the variable `::testing::GTEST_FLAG(death_test_style)` (which is initialized from the command-line flag `--gtest_death_test_style`).

- On POSIX systems, `fork()` (or `clone()` on Linux) is used to spawn the child, after which:
  - If the variable's value is `"fast"`, the death test statement is immediately executed.
  - If the variable's value is `"threadsafe"`, the child process re-executes the unit test binary just as it was originally invoked, but with some extra flags to cause just the single death test under consideration to be run.
- On Windows, the child is spawned using the `CreateProcess()` API, and re-executes the binary to cause just the single death test under consideration to be run - much like the `threadsafe` mode on POSIX.

Other values for the variable are illegal and will cause the death test to fail. Currently, the flag's default value is **"fast"**

1. the child's exit status satisfies the predicate, and
2. the child's stderr matches the regular expression.

If the death test statement runs to completion without dying, the child process will nonetheless terminate, and the assertion fails.

#### 8.4.5. Death Tests And Threads

The reason for the two death test styles has to do with thread safety. Due to well-known problems with forking in the presence of threads, death tests should

be run in a single-threaded context. Sometimes, however, it isn't feasible to arrange that kind of environment. For example, statically-initialized modules may start threads before `main` is ever reached. Once threads have been created, it may be difficult or impossible to clean them up.

googletest has three features intended to raise awareness of threading issues.

1. A warning is emitted if multiple threads are running when a death test is encountered.
2. Test suites with a name ending in "DeathTest" are run before all other tests.
3. It uses `clone()` instead of `fork()` to spawn the child process on Linux (`clone()` is not available on Cygwin and Mac), as `fork()` is more likely to cause the child to hang when the parent process has multiple threads.

It's perfectly fine to create threads inside a death test statement; they are executed in a separate process and cannot affect the parent.

#### 8.4.6. Death Test Styles

The "threadsafe" death test style was introduced in order to help mitigate the risks of testing in a possibly multithreaded environment. It trades increased test execution time (potentially dramatically so) for improved thread safety.

The automated testing framework does not set the style flag. You can choose a particular style of death tests by setting the flag programmatically:

```
testing::FLAGS_gtest_death_test_style="threadsafe"
```

You can do this in `main()` to set the style for all death tests in the binary, or in individual tests. Recall that flags are saved before running each test and restored afterwards, so you need not do that yourself. For example:

```
int main(int argc, char** argv) {
    testing::InitGoogleTest(&argc, argv);
    testing::FLAGS_gtest_death_test_style = "fast";
    return RUN_ALL_TESTS();
}

TEST(MyDeathTest, TestOne) {
    testing::FLAGS_gtest_death_test_style = "threadsafe";
    // This test is run in the "threadsafe" style:
    ASSERT_DEATH(ThisShouldDie(), "");
}

TEST(MyDeathTest, TestTwo) {
    // This test is run in the "fast" style:
    ASSERT_DEATH(ThisShouldDie(), "");
}
```

#### 8.4.7. Caveats

The `statement` argument of `ASSERT_EXIT()` can be any valid C++ statement. If it leaves the current function via a `return` statement or by throwing an

exception, the death test is considered to have failed. Some googletest macros may return from the current function (e.g. `ASSERT_TRUE()`), so be sure to avoid them in `statement`.

Since `statement` runs in the child process, any in-memory side effect (e.g. modifying a variable, releasing memory, etc) it causes will *not* be observable in the parent process. In particular, if you release memory in a death test, your program will fail the heap check as the parent process will never see the memory reclaimed. To solve this problem, you can

1. try not to free memory in a death test;
2. free the memory again in the parent process; or
3. do not use the heap checker in your program.

Due to an implementation detail, you cannot place multiple death test assertions on the same line; otherwise, compilation will fail with an unobvious error message.

Despite the improved thread safety afforded by the “threadsafe” style of death test, thread problems such as deadlock are still possible in the presence of handlers registered with `pthread_atfork(3)`.

## 8.5. Using Assertions in Sub-routines

Note: If you want to put a series of test assertions in a subroutine to check for a complex condition, consider using a custom GMock matcher instead. This lets you provide a more readable error message in case of failure and avoid all of the issues described below.

### 8.5.1. Adding Traces to Assertions

If a test sub-routine is called from several places, when an assertion inside it fails, it can be hard to tell which invocation of the sub-routine the failure is from. You can alleviate this problem using extra logging or custom failure messages, but that usually clutters up your tests. A better solution is to use the `SCOPED_TRACE` macro or the `ScopedTrace` utility:

```
SCOPED_TRACE(message);  
  
ScopedTrace trace("file_path", line_number, message);
```

where `message` can be anything streamable to `std::ostream`. `SCOPED_TRACE` macro will cause the current file name, line number, and the given message to be added in every failure message. `ScopedTrace` accepts explicit file name and line number in arguments, which is useful for writing test helpers. The effect will be undone when the control leaves the current lexical scope.

For example,

```
10: void Sub1(int n) {  
11:     EXPECT_EQ(Bar(n), 1);  
12:     EXPECT_EQ(Bar(n + 1), 2);  
13: }  
14:
```



```
15: TEST(FooTest, Bar) {
16:     {
17:         SCOPED_TRACE("A"); // This trace point will be included in
18:                             // every failure in this scope.
19:         Sub1(1);
20:     }
21:     // Now it won't.
22:     Sub1(9);
23: }
```

could result in messages like these:

```
path/to/foo_test.cc:11: Failure
Value of: Bar(n)
Expected: 1
Actual: 2
Google Test trace:
path/to/foo_test.cc:17: A
```

```
path/to/foo_test.cc:12: Failure
Value of: Bar(n + 1)
Expected: 2
Actual: 3
```

Without the trace, it would've been difficult to know which invocation of `Sub1()` the two failures come from respectively. (You could add an extra message to each assertion in `Sub1()` to indicate the value of `n`, but that's tedious.)

Some tips on using `SCOPED_TRACE`:

1. With a suitable message, it's often enough to use `SCOPED_TRACE` at the beginning of a sub-routine, instead of at each call site.
2. When calling sub-routines inside a loop, make the loop iterator part of the message in `SCOPED_TRACE` such that you can know which iteration the failure is from.
3. Sometimes the line number of the trace point is enough for identifying the particular invocation of a sub-routine. In this case, you don't have to choose a unique message for `SCOPED_TRACE`. You can simply use `.`
4. You can use `SCOPED_TRACE` in an inner scope when there is one in the outer scope. In this case, all active trace points will be included in the failure messages, in reverse order they are encountered.
5. The trace dump is clickable in Emacs - hit **return** on a line number and you'll be taken to that line in the source file!

### 8.5.2. Propagating Fatal Failures

A common pitfall when using `ASSERT_*` and `FAIL*` is not understanding that when they fail they only abort the *current function*, not the entire test. For example, the following test will segfault:

```
void Subroutine() {
    // Generates a fatal failure and aborts the current function.
```



```
ASSERT_EQ(1, 2);

// The following won't be executed.
...
}

TEST(FooTest, Bar) {
    Subroutine(); // The intended behavior is for the fatal failure
                  // in Subroutine() to abort the entire test.

    // The actual behavior: the function goes on after Subroutine() returns.
    int* p = nullptr;
    *p = 3; // Segfault!
}
```

To alleviate this, googletest provides three different solutions. You could use either exceptions, the (ASSERT|EXPECT)\_NO\_FATAL\_FAILURE assertions or the HasFatalFailure() function. They are described in the following two subsections.

**Asserting on Subroutines with an exception** The following code can turn ASSERT-failure into an exception:

```
class ThrowListener : public testing::EmptyTestEventListener {
    void OnTestPartResult(const testing::TestPartResult& result) override {
        if (result.type() == testing::TestPartResult::kFatalFailure) {
            throw testing::AssertionException(result);
        }
    }
};

int main(int argc, char** argv) {
    ...
    testing::UnitTest::GetInstance()->listeners().Append(new ThrowListener);
    return RUN_ALL_TESTS();
}
```

This listener should be added after other listeners if you have any, otherwise they won't see failed OnTestPartResult.

**Asserting on Subroutines** As shown above, if your test calls a subroutine that has an ASSERT\_\* failure in it, the test will continue after the subroutine returns. This may not be what you want.

Often people want fatal failures to propagate like exceptions. For that google-test offers the following macros:



Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_NO_FATAL_FAILURE(statement)</code>	<code>EXPECT_NO_FATAL_FAILURE(statement)</code>	<code>statement</code> doesn't generate any new fatal failures in the current thread.

Only failures in the thread that executes the assertion are checked to determine the result of this type of assertions. If `statement` creates new threads, failures in these threads are ignored.

Examples:

```
ASSERT_NO_FATAL_FAILURE(Foo());
```

```
int i;
EXPECT_NO_FATAL_FAILURE({
    i = Bar();
});
```

Assertions from multiple threads are currently not supported on Windows.

**Checking for Failures in the Current Test** `HasFatalFailure()` in the `::testing::Test` class returns `true` if an assertion in the current test has suffered a fatal failure. This allows functions to catch fatal failures in a sub-routine and return early.

```
class Test {
public:
    ...
    static bool HasFatalFailure();
};
```

The typical usage, which basically simulates the behavior of a thrown exception, is:

```
TEST(FooTest, Bar) {
    Subroutine();
    // Aborts if Subroutine() had a fatal failure.
    if (HasFatalFailure()) return;

    // The following won't be executed.
    ...
}
```

If `HasFatalFailure()` is used outside of `TEST()` , `TEST_F()` , or a test fixture,

you must add the `::testing::Test::` prefix, as in:

```
if (testing::Test::HasFatalFailure()) return;
```

Similarly, `HasNonfatalFailure()` returns `true` if the current test has at least one non-fatal failure, and `HasFailure()` returns `true` if the current test has at least one failure of either kind.

## 8.6. Logging Additional Information

In your test code, you can call `RecordProperty("key", value)` to log additional information, where `value` can be either a string or an `int`. The *last* value recorded for a key will be emitted to the XML output if you specify one. For example, the test

```
TEST_F(WidgetUsageTest, MinAndMaxWidgets) {  
    RecordProperty("MaximumWidgets", ComputeMaxUsage());  
    RecordProperty("MinimumWidgets", ComputeMinUsage());  
}
```

will output XML like this:

```
...  
<testcase name="MinAndMaxWidgets" status="run" time="0.006" classname="WidgetUsage...  
...
```

NOTE:

- `RecordProperty()` is a static member of the `Test` class. Therefore it needs to be prefixed with `::testing::Test::` if used outside of the `TEST` body and the test fixture class.
- `key` must be a valid XML attribute name, and cannot conflict with the ones already used by googletest (`name`, `status`, `time`, `classname`, `type_param`, and `value_param`).
- Calling `RecordProperty()` outside of the lifespan of a test is allowed. If it's called outside of a test but between a test suite's `SetUpTestSuite()` and `TearDownTestSuite()` methods, it will be attributed to the XML element for the test suite. If it's called outside of all test suites (e.g. in a test environment), it will be attributed to the top-level XML element.

## 8.7. Sharing Resources Between Tests in the Same Test Suite

googletest creates a new test fixture object for each test in order to make tests independent and easier to debug. However, sometimes tests use resources that are expensive to set up, making the one-copy-per-test model prohibitively expensive.

If the tests don't change the resource, there's no harm in their sharing a single resource copy. So, in addition to per-test set-up/tear-down, googletest also supports per-test-suite set-up/tear-down. To use it:

1. In your test fixture class (say `FooTest` ), declare as `static` some member variables to hold the shared resources.
2. Outside your test fixture class (typically just below it), define those member variables, optionally giving them initial values.
3. In the same test fixture class, define a `static void SetUpTestSuite()` function (remember not to spell it as `SetupTestSuite` with a small `u`!) to set up the shared resources and a `static void TearDownTestSuite()` function to tear them down.

That's it! googletest automatically calls `SetUpTestSuite()` before running the *first test* in the `FooTest` test suite (i.e. before creating the first `FooTest` object), and calls `TearDownTestSuite()` after running the *last test* in it (i.e. after deleting the last `FooTest` object). In between, the tests can use the shared resources.

Remember that the test order is undefined, so your code can't depend on a test preceding or following another. Also, the tests must either not modify the state of any shared resource, or, if they do modify the state, they must restore the state to its original value before passing control to the next test.

Here's an example of per-test-suite set-up and tear-down:

```
class FooTest : public testing::Test {
protected:
    // Per-test-suite set-up.
    // Called before the first test in this test suite.
    // Can be omitted if not needed.
    static void SetUpTestSuite() {
        shared_resource_ = new ...;
    }

    // Per-test-suite tear-down.
    // Called after the last test in this test suite.
    // Can be omitted if not needed.
    static void TearDownTestSuite() {
        delete shared_resource_;
        shared_resource_ = nullptr;
    }

    // You can define per-test set-up logic as usual.
    virtual void SetUp() { ... }

    // You can define per-test tear-down logic as usual.
    virtual void TearDown() { ... }

    // Some expensive resource shared by all tests.
    static T* shared_resource_;
};

T* FooTest::shared_resource_ = nullptr;

TEST_F(FooTest, Test1) {
    ... you can refer to shared_resource_ here ...
}
```



```

}

TEST_F(FooTest, Test2) {
    ... you can refer to shared_resource_ here ...
}

```

NOTE: Though the above code declares `SetUpTestSuite()` protected, it may sometimes be necessary to declare it public, such as when using it with `TEST_P`.

## 8.8. Global Set-Up and Tear-Down

Just as you can do set-up and tear-down at the test level and the test suite level, you can also do it at the test program level. Here's how.

First, you subclass the `::testing::Environment` class to define a test environment, which knows how to set-up and tear-down:

```

class Environment : public testing::Environment {
public:
    ~Environment() override {}

    // Override this to define how to set up the environment.
    void SetUp() override {}

    // Override this to define how to tear down the environment.
    void TearDown() override {}
};

```

Then, you register an instance of your environment class with googletest by calling the `::testing::AddGlobalTestEnvironment()` function:

```
Environment* AddGlobalTestEnvironment(Environment* env);
```

Now, when `RUN_ALL_TESTS()` is called, it first calls the `SetUp()` method of each environment object, then runs the tests if none of the environments reported fatal failures and `GTEST_SKIP()` was not called. `RUN_ALL_TESTS()` always calls `TearDown()` with each environment object, regardless of whether or not the tests were run.

It's OK to register multiple environment objects. In this suite, their `SetUp()` will be called in the order they are registered, and their `TearDown()` will be called in the reverse order.

Note that googletest takes ownership of the registered environment objects. Therefore **do not delete them** by yourself.

You should call `AddGlobalTestEnvironment()` before `RUN_ALL_TESTS()` is called, probably in `main()`. If you use `gtest_main`, you need to call this before `main()` starts for it to take effect. One way to do this is to define a global variable like this:

```

testing::Environment* const foo_env =
    testing::AddGlobalTestEnvironment(new FooEnvironment);

```

However, we strongly recommend you to write your own `main()` and call `AddGlobalTestEnvironment()` there, as relying on initialization of global variables makes the code harder to read and may cause problems when you register multiple environments from different translation units and the environments have dependencies among them (remember that the compiler doesn't guarantee the order in which global variables from different translation units are initialized).

## 8.9. Value-Parameterized Tests

*Value-parameterized tests* allow you to test your code with different parameters without writing multiple copies of the same test. This is useful in a number of situations, for example:

- You have a piece of code whose behavior is affected by one or more command-line flags. You want to make sure your code performs correctly for various values of those flags.
- You want to test different implementations of an OO interface.
- You want to test your code over various inputs (a.k.a. data-driven testing). This feature is easy to abuse, so please exercise your good sense when doing it!

### 8.9.1. How to Write Value-Parameterized Tests

To write value-parameterized tests, first you should define a fixture class. It must be derived from both `testing::Test` and `testing::WithParamInterface<T>` (the latter is a pure interface), where `T` is the type of your parameter values. For convenience, you can just derive the fixture class from `testing::TestWithParam<T>`, which itself is derived from both `testing::Test` and `testing::WithParamInterface<T>`. `T` can be any copyable type. If it's a raw pointer, you are responsible for managing the lifespan of the pointed values.

NOTE: If your test fixture defines `SetUpTestSuite()` or `TearDownTestSuite()` they must be declared **public** rather than **protected** in order to use `TEST_P`.

```
class FooTest :
    public testing::TestWithParam<const char*> {
    // You can implement all the usual fixture class members here.
    // To access the test parameter, call GetParam() from class
    // TestWithParam<T>.
};

// Or, when you want to add parameters to a pre-existing fixture class:
class BaseTest : public testing::Test {
    ...
};
class BarTest : public BaseTest,
                public testing::WithParamInterface<const char*> {
    ...
};
```



Then, use the `TEST_P` macro to define as many test patterns using this fixture as you want. The `_P` suffix is for “parameterized” or “pattern”, whichever you prefer to think.

```
TEST_P(FooTest, DoesBlah) {  
    // Inside a test, access the test parameter with the GetParam() method  
    // of the TestWithParam<T> class:  
    EXPECT_TRUE(foo.Blah(GetParam()));  
    ...  
}  
  
TEST_P(FooTest, HasBlahBlah) {  
    ...  
}
```

Finally, you can use `INstantiate_Test_Suite_P` to instantiate the test suite with any set of parameters you want. googletest defines a number of functions for generating test parameters. They return what we call (surprise!) *parameter generators*. Here is a summary of them, which are all in the `testing` namespace:

Parameter Generator	Behavior
<code>Range(begin, end [, step])</code>	Yields values {begin, begin+step, begin+step+step, ...}. The values do not include end. step defaults to 1.
<code>Values(v1, v2, ..., vN)</code>	Yields values {v1, v2, ..., vN}.
<code>ValuesIn(container)</code> and <code>ValuesIn(begin, end)</code>	Yields values from a C-style array, an STL-style container, or an iterator range [begin, end)
<code>Bool()</code>	Yields sequence {false, true}.
<code>Combine(g1, g2, ..., gN)</code>	Yields all combinations (Cartesian product) as std::tuples of the values generated by the N generators.

For more details, see the comments at the definitions of these functions.

The following statement will instantiate tests from the `FooTest` test suite each with parameter values "meeny", "miny", and "moe".

```
INstantiate_Test_Suite_P(InstantiationName,  
                        FooTest,  
                        testing::Values("meeny", "miny", "moe"));
```

NOTE: The code above must be placed at global or namespace scope, not at function scope.

Per default, every `TEST_P` without a corresponding `INstantiate_Test_Suite_P` causes a failing test in test suite `GoogleTestVerification`. If you have a test suite where that omission is not an error, for example it is in a library

that may be linked in for other reason or where the list of test cases is dynamic and may be empty, then this check can be suppressed by tagging the test suite:

```
GTEST_ALLOW_UNINSTANTIATED_PARAMETERIZED_TEST(FooTest);
```

To distinguish different instances of the pattern (yes, you can instantiate it more than once), the first argument to `INstantiateTestSuite_P` is a prefix that will be added to the actual test suite name. Remember to pick unique prefixes for different instantiations. The tests from the instantiation above will have these names:

- `InstantiationName/FooTest.DoesBlah/0` for "meeny"
- `InstantiationName/FooTest.DoesBlah/1` for "miny"
- `InstantiationName/FooTest.DoesBlah/2` for "moe"
- `InstantiationName/FooTest.HasBlahBlah/0` for "meeny"
- `InstantiationName/FooTest.HasBlahBlah/1` for "miny"
- `InstantiationName/FooTest.HasBlahBlah/2` for "moe"

You can use these names in `--gtest_filter`.

This statement will instantiate all tests from `FooTest` again, each with parameter values `cat` and `dog`:

```
const char* pets[] = {"cat", "dog"};  
INstantiateTestSuite_P(AnotherInstantiationName, FooTest,  
                      testing::ValuesIn(pets));
```

The tests from the instantiation above will have these names:

- `AnotherInstantiationName/FooTest.DoesBlah/0` for `cat`
- `AnotherInstantiationName/FooTest.DoesBlah/1` for "dog"
- `AnotherInstantiationName/FooTest.HasBlahBlah/0` for `cat`
- `AnotherInstantiationName/FooTest.HasBlahBlah/1` for "dog"

Please note that `INstantiateTestSuite_P` will instantiate *all* tests in the given test suite, whether their definitions come before or *after* the `INstantiateTestSuite_P` statement.

You can see `sample7_unittest.cc` and `sample8_unittest.cc` for more examples.

### 8.9.2. Creating Value-Parameterized Abstract Tests

In the above, we define and instantiate `FooTest` in the *same* source file. Sometimes you may want to define value-parameterized tests in a library and let other people instantiate them later. This pattern is known as *abstract tests*. As an example of its application, when you are designing an interface you can write a standard suite of abstract tests (perhaps using a factory function as the test parameter) that all implementations of the interface are expected to pass. When someone implements the interface, they can instantiate your suite to get all the interface-conformance tests for free.

To define abstract tests, you should organize your code like this:

1. Put the definition of the parameterized test fixture class (e.g. `FooTest`)

in a header file, say `foo_param_test.h`. Think of this as *declaring* your abstract tests.

2. Put the `TEST_P` definitions in `foo_param_test.cc`, which includes `foo_param_test.h`. Think of this as *implementing* your abstract tests.

Once they are defined, you can instantiate them by including `foo_param_test.h`, invoking `INstantiateTestSuiteP()`, and depending on the library target that contains `foo_param_test.cc`. You can instantiate the same abstract test suite multiple times, possibly in different source files.

### 8.9.3. Specifying Names for Value-Parameterized Test Parameters

The optional last argument to `INstantiateTestSuiteP()` allows the user to specify a function or functor that generates custom test name suffixes based on the test parameters. The function should accept one argument of type `testing::TestParamInfo<class ParamType>`, and return `std::string`.

`testing::PrintToStringParamName` is a builtin test suffix generator that returns the value of `testing::PrintToString(GetParam())`. It does not work for `std::string` or C strings.

NOTE: test names must be non-empty, unique, and may only contain ASCII alphanumeric characters. In particular, they should not contain underscores

```
class MyTestSuite : public testing::TestWithParam<int> {};
```

```
TEST_P(MyTestSuite, MyTest)
{
    std::cout << "Example Test Param: " << GetParam() << std::endl;
}
```

```
INstantiateTestSuiteP(MyGroup, MyTestSuite, testing::Range(0, 10),
    testing::PrintToStringParamName());
```

Providing a custom functor allows for more control over test parameter name generation, especially for types where the automatic conversion does not generate helpful parameter names (e.g. strings as demonstrated above). The following example illustrates this for multiple parameters, an enumeration type and a string, and also demonstrates how to combine generators. It uses a lambda for conciseness:

```
enum class MyType { MY_FOO = 0, MY_BAR = 1 };
```

```
class MyTestSuite : public testing::TestWithParam<std::tuple<MyType, std::string>> {
};
```

```
INstantiateTestSuiteP(
    MyGroup, MyTestSuite,
    testing::Combine(
        testing::Values(MyType::VALUE_0, MyType::VALUE_1),
        testing::ValuesIn("", "")),
    [](const testing::TestParamInfo<MyTestSuite::ParamType>& info) {
```



```
std::string name = absl::StrCat(
    std::get<0>(info.param) == MY_FOO ? "Foo" : "Bar", "_",
    std::get<1>(info.param));
absl::c_replace_if(name, [](char c) { return !std::isalnum(c); }, '_');
return name;
});
```

## 8.10. Typed Tests

Suppose you have multiple implementations of the same interface and want to make sure that all of them satisfy some common requirements. Or, you may have defined several types that are supposed to conform to the same “concept” and you want to verify it. In both cases, you want the same test logic repeated for different types.

While you can write one `TEST` or `TEST_F` for each type you want to test (and you may even factor the test logic into a function template that you invoke from the `TEST`), it’s tedious and doesn’t scale: if you want `m` tests over `n` types, you’ll end up writing `m*n` `TEST`s.

*Typed tests* allow you to repeat the same test logic over a list of types. You only need to write the test logic once, although you must know the type list when writing typed tests. Here’s how you do it:

First, define a fixture class template. It should be parameterized by a type. Remember to derive it from `::testing::Test`:

```
template <typename T>
class FooTest : public testing::Test {
public:
    ...
    using List = std::list<T>;
    static T shared_;
    T value_;
};
```

Next, associate a list of types with the test suite, which will be repeated for each type in the list:

```
using MyTypes = ::testing::Types<char, int, unsigned int>;
TYPED_TEST_SUITE(FooTest, MyTypes);
```

The type alias (`using` or `typedef`) is necessary for the `TYPED_TEST_SUITE` macro to parse correctly. Otherwise the compiler will think that each comma in the type list introduces a new macro argument.

Then, use `TYPED_TEST()` instead of `TEST_F()` to define a typed test for this test suite. You can repeat this as many times as you want:

```
TYPED_TEST(FooTest, DoesBlah) {
    // Inside a test, refer to the special name TypeParam to get the type
    // parameter. Since we are inside a derived class template, C++ requires
    // us to visit the members of FooTest via 'this'.
    TypeParam n = this->value_;
```



```
// To visit static members of the fixture, add the 'TestFixture::'
// prefix.
n += TestFixture::shared_;

// To refer to typedefs in the fixture, add the 'typename TestFixture::'
// prefix. The 'typename' is required to satisfy the compiler.
typename TestFixture::List values;

values.push_back(n);
...
}

TYPED_TEST(FooTest, HasPropertyA) { ... }
```

You can see `sample6_unittest.cc` for a complete example.

## 8.11. Type-Parameterized Tests

*Type-parameterized tests* are like typed tests, except that they don't require you to know the list of types ahead of time. Instead, you can define the test logic first and instantiate it with different type lists later. You can even instantiate it more than once in the same program.

If you are designing an interface or concept, you can define a suite of type-parameterized tests to verify properties that any valid implementation of the interface/concept should have. Then, the author of each implementation can just instantiate the test suite with their type to verify that it conforms to the requirements, without having to write similar tests repeatedly. Here's an example:

First, define a fixture class template, as we did with typed tests:

```
template <typename T>
class FooTest : public testing::Test {
    ...
};
```

Next, declare that you will define a type-parameterized test suite:

```
TYPED_TEST_SUITE_P(FooTest);
```

Then, use `TYPED_TEST_P()` to define a type-parameterized test. You can repeat this as many times as you want:

```
TYPED_TEST_P(FooTest, DoesBlah) {
    // Inside a test, refer to TypeParam to get the type parameter.
    TypeParam n = 0;
    ...
}
```

```
TYPED_TEST_P(FooTest, HasPropertyA) { ... }
```

Now the tricky part: you need to register all test patterns using the `REGISTER_TYPED_TEST_SUITE_P` macro before you can instantiate them. The first argument of the macro is the test suite name; the rest are the names of

the tests in this test suite:

```
REGISTER_TYPED_TEST_SUITE_P(FooTest,  
                             DoesBlah, HasPropertyA);
```

Finally, you are free to instantiate the pattern with the types you want. If you put the above code in a header file, you can `#include` it in multiple C++ source files and instantiate it multiple times.

```
using MyTypes = ::testing::Types<char, int, unsigned int>;  
INSTANTIATE_TYPED_TEST_SUITE_P(My, FooTest, MyTypes);
```

To distinguish different instances of the pattern, the first argument to the `INSTANTIATE_TYPED_TEST_SUITE_P` macro is a prefix that will be added to the actual test suite name. Remember to pick unique prefixes for different instances.

In the special case where the type list contains only one type, you can write that type directly without `::testing::Types<...>`, like this:

```
INSTANTIATE_TYPED_TEST_SUITE_P(My, FooTest, int);
```

You can see `sample6_unittest.cc` for a complete example.

## 8.12. Testing Private Code

If you change your software's internal implementation, your tests should not break as long as the change is not observable by users. Therefore, **per the black-box testing principle, most of the time you should test your code through its public interfaces.**

**If you still find yourself needing to test internal implementation code, consider if there's a better design.** The desire to test internal implementation is often a sign that the class is doing too much. Consider extracting an implementation class, and testing it. Then use that implementation class in the original class.

If you absolutely have to test non-public interface code though, you can. There are two cases to consider:

- Static functions ( *not* the same as static member functions!) or unnamed namespaces, and
- Private or protected class members

To test them, we use the following special techniques:

- Both static functions and definitions/declarations in an unnamed namespace are only visible within the same translation unit. To test them, you can `#include` the entire `.cc` file being tested in your `*_test.cc` file. (`#including .cc` files is not a good way to reuse code - you should not do this in production code!)

However, a better approach is to move the private code into the `foo::internal` namespace, where `foo` is the namespace your project normally uses, and put the private declarations in a `*-internal.h` file. Your production `.cc` files and your tests are allowed to include this



internal header, but your clients are not. This way, you can fully test your internal implementation without leaking it to your clients.

- Private class members are only accessible from within the class or by friends. To access a class' private members, you can declare your test fixture as a friend to the class and define accessors in your fixture. Tests using the fixture can then access the private members of your production class via the accessors in the fixture. Note that even though your fixture is a friend to your production class, your tests are not automatically friends to it, as they are technically defined in sub-classes of the fixture.

Another way to test private members is to refactor them into an implementation class, which is then declared in a *\*-internal.h* file. Your clients aren't allowed to include this header but your tests can. Such is called the Pimpl (Private Implementation) idiom.

Or, you can declare an individual test as a friend of your class by adding this line in the class body:

```
FRIEND_TEST(TestSuiteName, TestName);
```

For example,

```
// foo.h
class Foo {
    ...
private:
    FRIEND_TEST(FooTest, BarReturnsZeroOnNull);

    int Bar(void* x);
};

// foo_test.cc
...
TEST(FooTest, BarReturnsZeroOnNull) {
    Foo foo;
    EXPECT_EQ(foo.Bar(NULL), 0); // Uses Foo's private member Bar().
}
```

Pay special attention when your class is defined in a namespace, as you should define your test fixtures and tests in the same namespace if you want them to be friends of your class. For example, if the code to be tested looks like:

```
namespace my_namespace {

class Foo {
    friend class FooTest;
    FRIEND_TEST(FooTest, Bar);
    FRIEND_TEST(FooTest, Baz);
    ... definition of the class Foo ...
};

} // namespace my_namespace
```

Your test code should be something like:

```
namespace my_namespace {

class FooTest : public testing::Test {
protected:
    ...
};

TEST_F(FooTest, Bar) { ... }
TEST_F(FooTest, Baz) { ... }

} // namespace my_namespace
```

### 8.13. “Catching” Failures

If you are building a testing utility on top of googletest, you’ll want to test your utility. What framework would you use to test it? googletest, of course.

The challenge is to verify that your testing utility reports failures correctly. In frameworks that report a failure by throwing an exception, you could catch the exception and assert on it. But googletest doesn’t use exceptions, so how do we test that a piece of code generates an expected failure?

"gtest/gtest-spi.h" contains some constructs to do this. After #including this header, you can use

```
EXPECT_FATAL_FAILURE(statement, substring);
```

to assert that `statement` generates a fatal (e.g. `ASSERT_*`) failure in the current thread whose message contains the given `substring`, or use

```
EXPECT_NONFATAL_FAILURE(statement, substring);
```

if you are expecting a non-fatal (e.g. `EXPECT_*`) failure.

Only failures in the current thread are checked to determine the result of this type of expectations. If `statement` creates new threads, failures in these threads are also ignored. If you want to catch failures in other threads as well, use one of the following macros instead:

```
EXPECT_FATAL_FAILURE_ON_ALL_THREADS(statement, substring);
EXPECT_NONFATAL_FAILURE_ON_ALL_THREADS(statement, substring);
```

NOTE: Assertions from multiple threads are currently not supported on Windows.

For technical reasons, there are some caveats:

1. You cannot stream a failure message to either macro.
2. `statement` in `EXPECT_FATAL_FAILURE{_ON_ALL_THREADS}()` cannot reference local non-static variables or non-static members of `this` object.
3. `statement` in `EXPECT_FATAL_FAILURE{_ON_ALL_THREADS}()` cannot return a value.

## 8.14. Registering tests programmatically

The TEST macros handle the vast majority of all use cases, but there are few where runtime registration logic is required. For those cases, the framework provides the `::testing::RegisterTest` that allows callers to register arbitrary tests dynamically.

This is an advanced API only to be used when the TEST macros are insufficient. The macros should be preferred when possible, as they avoid most of the complexity of calling this function.

It provides the following signature:

```
template <typename Factory>
TestInfo* RegisterTest(const char* test_suite_name, const char* test_name,
                      const char* type_param, const char* value_param,
                      const char* file, int line, Factory factory);
```

The factory argument is a factory callable (move-constructible) object or function pointer that creates a new instance of the Test object. It handles ownership to the caller. The signature of the callable is `Fixture*()`, where `Fixture` is the test fixture class for the test. All tests registered with the same `test_suite_name` must return the same fixture type. This is checked at runtime.

The framework will infer the fixture class from the factory and will call the `SetUpTestSuite` and `TearDownTestSuite` for it.

Must be called before `RUN_ALL_TESTS()` is invoked, otherwise behavior is undefined.

Use case example:

```
class MyFixture : public testing::Test {
public:
    // All of these optional, just like in regular macro usage.
    static void SetUpTestSuite() { ... }
    static void TearDownTestSuite() { ... }
    void SetUp() override { ... }
    void TearDown() override { ... }
};

class MyTest : public MyFixture {
public:
    explicit MyTest(int data) : data_(data) {}
    void TestBody() override { ... }

private:
    int data_;
};

void RegisterMyTests(const std::vector<int>& values) {
    for (int v : values) {
        testing::RegisterTest(
            "MyFixture", ("Test" + std::to_string(v)).c_str(), nullptr,
```

```

        std::to_string(v).c_str(),
        __FILE__, __LINE__,
        // Important to use the fixture type as the return type here.
        [=]() -> MyFixture* { return new MyTest(v); });
    }
}
...
int main(int argc, char** argv) {
    std::vector<int> values_to_test = LoadValuesFromConfig();
    RegisterMyTests(values_to_test);
    ...
    return RUN_ALL_TESTS();
}

```

## 8.15. Getting the Current Test's Name

Sometimes a function may need to know the name of the currently running test. For example, you may be using the `SetUp()` method of your test fixture to set the golden file name based on which test is running. The `::testing::TestInfo` class has this information:

```

namespace testing {

class TestInfo {
public:
    // Returns the test suite name and the test name, respectively.
    //
    // Do NOT delete or free the return value - it's managed by the
    // TestInfo class.
    const char* test_suite_name() const;
    const char* name() const;
};
}

```

To obtain a `TestInfo` object for the currently running test, call `current_test_info()` on the `UnitTest` singleton object:

```

// Gets information about the currently running test.
// Do NOT delete the returned object - it's managed by the UnitTest class.
const testing::TestInfo* const test_info =
    testing::UnitTest::GetInstance()->current_test_info();

printf("We are in test %s of test suite %s.\n",
       test_info->name(),
       test_info->test_suite_name());

```

`current_test_info()` returns a null pointer if no test is running. In particular, you cannot find the test suite name in `SetUpTestSuite()`, `TearDownTestSuite()` (where you know the test suite name implicitly), or functions called from them.

## 8.16. Extending googletest by Handling Test Events

googletest provides an **event listener API** to let you receive notifications about the progress of a test program and test failures. The events you can listen to include the start and end of the test program, a test suite, or a test method, among others. You may use this API to augment or replace the standard console output, replace the XML output, or provide a completely different form of output, such as a GUI or a database. You can also use test events as checkpoints to implement a resource leak checker, for example.

### 8.16.1. Defining Event Listeners

To define a event listener, you subclass either `testing::TestEventListener` or `testing::EmptyTestEventListener`. The former is an (abstract) interface, where *each pure virtual method can be overridden to handle a test event* (For example, when a test starts, the `OnTestStart()` method will be called.). The latter provides an empty implementation of all methods in the interface, such that a subclass only needs to override the methods it cares about.

When an event is fired, its context is passed to the handler function as an argument. The following argument types are used:

- `UnitTest` reflects the state of the entire test program,
- `TestSuite` has information about a test suite, which can contain one or more tests,
- `TestInfo` contains the state of a test, and
- `TestPartResult` represents the result of a test assertion.

An event handler function can examine the argument it receives to find out interesting information about the event and the test program's state.

Here's an example:

```
class MinimalistPrinter : public testing::EmptyTestEventListener {
    // Called before a test starts.
    virtual void OnTestStart(const testing::TestInfo& test_info) {
        printf("*** Test %s.%s starting.\n",
               test_info.test_suite_name(), test_info.name());
    }

    // Called after a failed assertion or a SUCCESS().
    virtual void OnTestPartResult(const testing::TestPartResult& test_part_result) {
        printf("%s in %s: %d\n%s\n",
               test_part_result.failed() ? "*** Failure" : "Success",
               test_part_result.file_name(),
               test_part_result.line_number(),
               test_part_result.summary());
    }

    // Called after a test ends.
    virtual void OnTestEnd(const testing::TestInfo& test_info) {
        printf("*** Test %s.%s ending.\n",
               test_info.test_suite_name(), test_info.name());
    }
};
```

```
    }
};
```

### 8.16.2. Using Event Listeners

To use the event listener you have defined, add an instance of it to the googletest event listener list (represented by class `TestEventListeners` - note the “s” at the end of the name) in your `main()` function, before calling `RUN_ALL_TESTS()`:

```
int main(int argc, char** argv) {
    testing::InitGoogleTest(&argc, argv);
    // Gets hold of the event listener list.
    testing::TestEventListeners& listeners =
        testing::UnitTest::GetInstance()->listeners();
    // Adds a listener to the end. googletest takes the ownership.
    listeners.Append(new MinimalistPrinter);
    return RUN_ALL_TESTS();
}
```

There’s only one problem: the default test result printer is still in effect, so its output will mingle with the output from your minimalist printer. To suppress the default printer, just release it from the event listener list and delete it. You can do so by adding one line:

```
...
delete listeners.Release(listeners.default_result_printer());
listeners.Append(new MinimalistPrinter);
return RUN_ALL_TESTS();
```

Now, sit back and enjoy a completely different output from your tests. For more details, see `sample9_unittest.cc`.

You may append more than one listener to the list. When an `On*Start()` or `OnTestPartResult()` event is fired, the listeners will receive it in the order they appear in the list (since new listeners are added to the end of the list, the default text printer and the default XML generator will receive the event first). An `On*End()` event will be received by the listeners in the *reverse* order. This allows output by listeners added later to be framed by output from listeners added earlier.

### 8.16.3. Generating Failures in Listeners

You may use failure-raising macros (`EXPECT_*()`, `ASSERT_*()`, `FAIL()`, etc) when processing an event. There are some restrictions:

1. You cannot generate any failure in `OnTestPartResult()` (otherwise it will cause `OnTestPartResult()` to be called recursively).
2. A listener that handles `OnTestPartResult()` is not allowed to generate any failure.

When you add listeners to the listener list, you should put listeners that handle `OnTestPartResult()` *before* listeners that can generate failures. This ensures that failures generated by the latter are attributed to the right test



by the former.

See `sample10_unittest.cc` for an example of a failure-raising listener.

## 8.17. Running Test Programs: Advanced Options

googletest test programs are ordinary executables. Once built, you can run them directly and affect their behavior via the following environment variables and/or command line flags. For the flags to work, your programs must call `::testing::InitGoogleTest()` before calling `RUN_ALL_TESTS()`.

To see a list of supported flags and their usage, please run your test program with the `--help` flag. You can also use `-h`, `/?`, or `/?` for short.

If an option is specified both by an environment variable and by a flag, the latter takes precedence.

### 8.17.1. Selecting Tests

**Listing Test Names** Sometimes it is necessary to list the available tests in a program before running them so that a filter may be applied if needed. Including the flag `--gtest_list_tests` overrides all other flags and lists tests in the following format:

```
TestSuite1.  
  TestName1  
  TestName2  
TestSuite2.  
  TestName
```

None of the tests listed are actually run if the flag is provided. There is no corresponding environment variable for this flag.

**Running a Subset of the Tests** By default, a googletest program runs all tests the user has defined. Sometimes, you want to run only a subset of the tests (e.g. for debugging or quickly verifying a change). If you set the `GTEST_FILTER` environment variable or the `--gtest_filter` flag to a filter string, googletest will only run the tests whose full names (in the form of `TestSuiteName.TestName`) match the filter.

The format of a filter is a `':'`-separated list of wildcard patterns (called the *positive patterns*) optionally followed by a `-` and another `':'`-separated pattern list (called the *negative patterns*). A test matches the filter if and only if it matches any of the positive patterns but does not match any of the negative patterns.

A pattern may contain `'*'` (matches any string) or `'?'` (matches any single character). For convenience, the filter `'*-NegativePatterns'` can be also written as `'-NegativePatterns'`.

For example:

- `./foo_test` Has no flag, and thus runs all its tests.



- `./foo_test --gtest_filter=*` Also runs everything, due to the single match-everything `*` value.
- `./foo_test --gtest_filter=FooTest.*` Runs everything in test suite `FooTest`.
- `./foo_test --gtest_filter=*Null*:~Constructor*` Runs any test whose full name contains either "Null" or "Constructor".
- `./foo_test --gtest_filter=~DeathTest.*` Runs all non-death tests.
- `./foo_test --gtest_filter=FooTest.*~FooTest.Bar` Runs everything in test suite `FooTest` except `FooTest.Bar`.
- `./foo_test --gtest_filter=FooTest.*:BarTest.*~FooTest.Bar:BarTest.Foo` Runs everything in test suite `FooTest` except `FooTest.Bar` and everything in test suite `BarTest` except `BarTest.Foo`.

**Stop test execution upon first failure** By default, a googletest program runs all tests the user has defined. In some cases (e.g. iterative test development & execution) it may be desirable stop test execution upon first failure (trading improved latency for completeness). If `GTEST_FAIL_FAST` environment variable or `--gtest_fail_fast` flag is set, the test runner will stop execution as soon as the first test failure is found.

**Temporarily Disabling Tests** If you have a broken test that you cannot fix right away, you can add the `DISABLED_` prefix to its name. This will exclude it from execution. This is better than commenting out the code or using `#if 0`, as disabled tests are still compiled (and thus won't rot).

If you need to disable all tests in a test suite, you can either add `DISABLED_` to the front of the name of each test, or alternatively add it to the front of the test suite name.

For example, the following tests won't be run by googletest, even though they will still be compiled:

```
// Tests that Foo does Abc.
TEST(FooTest, DISABLED_DoesAbc) { ... }

class DISABLED_BarTest : public testing::Test { ... };

// Tests that Bar does Xyz.
TEST_F(DISABLED_BarTest, DoesXyz) { ... }
```

NOTE: This feature should only be used for temporary pain-relief. You still have to fix the disabled tests at a later date. As a reminder, googletest will print a banner warning you if a test program contains any disabled tests.

TIP: You can easily count the number of disabled tests you have using `gsearch` and/or `grep`. This number can be used as a metric for improving your test quality.

**Temporarily Enabling Disabled Tests** To include disabled tests in test execution, just invoke the test program with the `--gtest_also_run_disabled_tests`





flag or set the `GTEST_ALSO_RUN_DISABLED_TESTS` environment variable to a value other than 0. You can combine this with the `--gtest_filter` flag to further select which disabled tests to run.

### 8.17.2. Repeating the Tests

Once in a while you'll run into a test whose result is hit-or-miss. Perhaps it will fail only 1 % of the time, making it rather hard to reproduce the bug under a debugger. This can be a major source of frustration.

The `--gtest_repeat` flag allows you to repeat all (or selected) test methods in a program many times. Hopefully, a flaky test will eventually fail and give you a chance to debug. Here's how to use it:

```
$ foo_test --gtest_repeat=1000
```

Repeat `foo_test` 1000 times and don't stop at failures.

```
$ foo_test --gtest_repeat=-1
```

A negative count means repeating forever.

```
$ foo_test --gtest_repeat=1000 --gtest_break_on_failure
```

Repeat `foo_test` 1000 times, stopping at the first failure. This is especially useful when running under a debugger: when the test fails, it will drop into the debugger and you can then inspect variables and stacks.

```
$ foo_test --gtest_repeat=1000 --gtest_filter=FooBar.*
```

Repeat the tests whose name matches the filter 1000 times.

If your test program contains global set-up/tear-down code, it will be repeated in each iteration as well, as the flakiness may be in it. You can also specify the repeat count by setting the `GTEST_REPEAT` environment variable.

### 8.17.3. Shuffling the Tests

You can specify the `--gtest_shuffle` flag (or set the `GTEST_SHUFFLE` environment variable to 1) to run the tests in a program in a random order. This helps to reveal bad dependencies between tests.

By default, googletest uses a random seed calculated from the current time. Therefore you'll get a different order every time. The console output includes the random seed value, such that you can reproduce an order-related test failure later. To specify the random seed explicitly, use the `--gtest_random_seed=SEED` flag (or set the `GTEST_RANDOM_SEED` environment variable), where `SEED` is an integer in the range `[0, 99999]`. The seed value 0 is special: it tells googletest to do the default behavior of calculating the seed from the current time.

If you combine this with `--gtest_repeat=N`, googletest will pick a different random seed and re-shuffle the tests in each iteration.



#### 8.17.4. Controlling Test Output

**Colored Terminal Output** googletest can use colors in its terminal output to make it easier to spot the important information:

```
... [-----] 1 test from FooTest [ RUN      ] FooTest.DoesAbc [   OK
] FooTest.DoesAbc [-----] 2 tests from BarTest [ RUN      ] Bar-
Test.HasXyzProperty [   OK ] BarTest.HasXyzProperty [ RUN      ]
BarTest.ReturnsTrueOnSuccess ... some error messages ... [ FAILED ]
BarTest.ReturnsTrueOnSuccess ... [=====] 30 tests from 14
test suites ran. [  PASSED ] 28 tests. [  FAILED ] 2 tests, listed below: [
FAILED ] BarTest.ReturnsTrueOnSuccess [  FAILED ] AnotherTest.DoesXyz
2 FAILED TESTS
```

You can set the `GTEST_COLOR` environment variable or the `--gtest_color` command line flag to `yes`, `no`, or `auto` (the default) to enable colors, disable colors, or let googletest decide. When the value is `auto`, googletest will use colors if and only if the output goes to a terminal and (on non-Windows platforms) the `TERM` environment variable is set to `xterm` or `xterm-color`.

**Suppressing test passes** By default, googletest prints 1 line of output for each test, indicating if it passed or failed. To show only test failures, run the test program with `--gtest_brief=1`, or set the `GTEST_BRIEF` environment variable to 1.

**Suppressing the Elapsed Time** By default, googletest prints the time it takes to run each test. To disable that, run the test program with the `--gtest_print_time=0` command line flag, or set the `GTEST_PRINT_TIME` environment variable to 0.

**Suppressing UTF-8 Text Output** In case of assertion failures, googletest prints expected and actual values of type `string` both as hex-encoded strings as well as in readable UTF-8 text if they contain valid non-ASCII UTF-8 characters. If you want to suppress the UTF-8 text because, for example, you don't have an UTF-8 compatible output medium, run the test program with `--gtest_print_utf8=0` or set the `GTEST_PRINT_UTF8` environment variable to 0.

**Generating an XML Report** googletest can emit a detailed XML report to a file in addition to its normal textual output. The report contains the duration of each test, and thus can help you identify slow tests.

To generate the XML report, set the `GTEST_OUTPUT` environment variable or the `--gtest_output` flag to the string `"xml:path_to_output_file"`, which will create the file at the given location. You can also just use the string `"xml"`, in which case the output can be found in the `test_detail.xml` file in the current directory.

If you specify a directory (for example, `"xml:output/directory/"` on Linux or `"xml:output\directory\"` on Windows), googletest will create the XML file in that directory, named after the test executable (e.g. `foo_test.xml` for

test program `foo_test` or `foo_test.exe`). If the file already exists (perhaps left over from a previous run), googletest will pick a different name (e.g. `foo_test_1.xml`) to avoid overwriting it.

The report is based on the `junitreport` Ant task. Since that format was originally intended for Java, a little interpretation is required to make it apply to googletest tests, as shown here:

```
<testsuites name="AllTests" ...>
  <testsuite name="test_case_name" ...>
    <testcase name="test_name" ...>
      <failure message="..." />
      <failure message="..." />
      <failure message="..." />
    </testcase>
  </testsuite>
</testsuites>
```

- The root `<testsuites>` element corresponds to the entire test program.
- `<testsuite>` elements correspond to googletest test suites.
- `<testcase>` elements correspond to googletest test functions.

For instance, the following program

```
TEST(MathTest, Addition) { ... }
TEST(MathTest, Subtraction) { ... }
TEST(LogicTest, NonContradiction) { ... }
```

could generate this report:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="3" failures="1" errors="0" time="0.035" timestamp="2011-10-31T18:52"
  <testsuite name="MathTest" tests="2" failures="1" errors="0" time="0.015">
    <testcase name="Addition" status="run" time="0.007" classname="">
      <failure message="Value of: add(1, 1)&#x0A; Actual: 3&#x0A;Expected: 2" type="">
      <failure message="Value of: add(1, -1)&#x0A; Actual: 1&#x0A;Expected: 0" type="">
    </testcase>
    <testcase name="Subtraction" status="run" time="0.005" classname="">
    </testcase>
  </testsuite>
  <testsuite name="LogicTest" tests="1" failures="0" errors="0" time="0.005">
    <testcase name="NonContradiction" status="run" time="0.005" classname="">
    </testcase>
  </testsuite>
</testsuites>
```

Things to note:

- The `tests` attribute of a `<testsuites>` or `<testsuite>` element tells how many test functions the googletest program or test suite contains, while the `failures` attribute tells how many of them failed.
- The `time` attribute expresses the duration of the test, test suite, or entire test program in seconds.
- The `timestamp` attribute records the local date and time of the test

execution.

- Each <failure> element corresponds to a single failed googletest assertion.

**Generating a JSON Report** googletest can also emit a JSON report as an alternative format to XML. To generate the JSON report, set the GTEST\_OUTPUT environment variable or the `--gtest_output` flag to the string `"json:path_to_output_file"`, which will create the file at the given location. You can also just use the string `"json"`, in which case the output can be found in the `test_detail.json` file in the current directory.

The report format conforms to the following JSON Schema:

```
{
  "$schema": "http://json-schema.org/schema#",
  "type": "object",
  "definitions": {
    "TestCase": {
      "type": "object",
      "properties": {
        "name": { "type": "string" },
        "tests": { "type": "integer" },
        "failures": { "type": "integer" },
        "disabled": { "type": "integer" },
        "time": { "type": "string" },
        "testsuite": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/TestInfo"
          }
        }
      }
    },
    "TestInfo": {
      "type": "object",
      "properties": {
        "name": { "type": "string" },
        "status": {
          "type": "string",
          "enum": ["RUN", "NOTRUN"]
        },
        "time": { "type": "string" },
        "classname": { "type": "string" },
        "failures": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/Failure"
          }
        }
      }
    },
    "Failure": {
```

```

        "type": "object",
        "properties": {
            "failures": { "type": "string" },
            "type": { "type": "string" }
        }
    },
    "properties": {
        "tests": { "type": "integer" },
        "failures": { "type": "integer" },
        "disabled": { "type": "integer" },
        "errors": { "type": "integer" },
        "timestamp": {
            "type": "string",
            "format": "date-time"
        },
        "time": { "type": "string" },
        "name": { "type": "string" },
        "testsuites": {
            "type": "array",
            "items": {
                "$ref": "#/definitions/TestCase"
            }
        }
    }
}

```

The report uses the format that conforms to the following Proto3 using the JSON encoding:

```

syntax = "proto3";

package googletest;

import "google/protobuf/timestamp.proto";
import "google/protobuf/duration.proto";

message UnitTest {
    int32 tests = 1;
    int32 failures = 2;
    int32 disabled = 3;
    int32 errors = 4;
    google.protobuf.Timestamp timestamp = 5;
    google.protobuf.Duration time = 6;
    string name = 7;
    repeated TestCase testsuites = 8;
}

message TestCase {
    string name = 1;
    int32 tests = 2;
    int32 failures = 3;
    int32 disabled = 4;
}

```

```

    int32 errors = 5;
    google.protobuf.Duration time = 6;
    repeated TestInfo testsuite = 7;
}

message TestInfo {
    string name = 1;
    enum Status {
        RUN = 0;
        NOTRUN = 1;
    }
    Status status = 2;
    google.protobuf.Duration time = 3;
    string classname = 4;
    message Failure {
        string failures = 1;
        string type = 2;
    }
    repeated Failure failures = 5;
}

```

For instance, the following program

```

TEST(MathTest, Addition) { ... }
TEST(MathTest, Subtraction) { ... }
TEST(LogicTest, NonContradiction) { ... }

```

could generate this report:

```

{
  "tests": 3,
  "failures": 1,
  "errors": 0,
  "time": "0.035s",
  "timestamp": "2011-10-31T18:52:42Z",
  "name": "AllTests",
  "testsuites": [
    {
      "name": "MathTest",
      "tests": 2,
      "failures": 1,
      "errors": 0,
      "time": "0.015s",
      "testsuite": [
        {
          "name": "Addition",
          "status": "RUN",
          "time": "0.007s",
          "classname": "",
          "failures": [
            {
              "message": "Value of: add(1, 1)\n Actual: 3\nExpected: 2",
              "type": ""
            }
          ],
        }
      ],
    }
  ],
}

```

```

        {
            "message": "Value of: add(1, -1)\n  Actual: 1\nExpected: 0",
            "type": ""
        }
    ]
},
{
    "name": "Subtraction",
    "status": "RUN",
    "time": "0.005s",
    "classname": ""
}
]
},
{
    "name": "LogicTest",
    "tests": 1,
    "failures": 0,
    "errors": 0,
    "time": "0.005s",
    "testsuite": [
        {
            "name": "NonContradiction",
            "status": "RUN",
            "time": "0.005s",
            "classname": ""
        }
    ]
}
]
}

```

IMPORTANT: The exact format of the JSON document is subject to change.

#### 8.17.5. Controlling How Failures Are Reported

**Detecting Test Premature Exit** Google Test implements the *premature-exit-file* protocol for test runners to catch any kind of unexpected exits of test programs. Upon start, Google Test creates the file which will be automatically deleted after all work has been finished. Then, the test runner can check if this file exists. In case the file remains undeleted, the inspected test has exited prematurely.

This feature is enabled only if the `TEST_PREMATURE_EXIT_FILE` environment variable has been set.

**Turning Assertion Failures into Break-Points** When running test programs under a debugger, it's very convenient if the debugger can catch an assertion failure and automatically drop into interactive mode. googletest's *break-on-failure* mode supports this behavior.

To enable it, set the `GTEST_BREAK_ON_FAILURE` environment variable to a va-



value other than 0. Alternatively, you can use the `--gtest_break_on_failure` command line flag.

**Disabling Catching Test-Thrown Exceptions** googletest can be used either with or without exceptions enabled. If a test throws a C++ exception or (on Windows) a structured exception (SEH), by default googletest catches it, reports it as a test failure, and continues with the next test method. This maximizes the coverage of a test run. Also, on Windows an uncaught exception will cause a pop-up window, so catching the exceptions allows you to run the tests automatically.

When debugging the test failures, however, you may instead want the exceptions to be handled by the debugger, such that you can examine the call stack when an exception is thrown. To achieve that, set the `GTEST_CATCH_EXCEPTIONS` environment variable to 0, or use the `--gtest_catch_exceptions=0` flag when running the tests.

#### 8.17.6. Sanitizer Integration

The Undefined Behavior Sanitizer, Address Sanitizer, and Thread Sanitizer all provide weak functions that you can override to trigger explicit failures when they detect sanitizer errors, such as creating a reference from `nullptr`. To override these functions, place definitions for them in a source file that you compile as part of your main binary:

```
extern "C" {
void __ubsan_on_report() {
    FAIL() << "Encountered an undefined behavior sanitizer error";
}
void __asan_on_error() {
    FAIL() << "Encountered an address sanitizer error";
}
void __tsan_on_report() {
    FAIL() << "Encountered a thread sanitizer error";
}
} // extern "C"
```

After compiling your project with one of the sanitizers enabled, if a particular test triggers a sanitizer error, googletest will report that it failed.



## 9. Código de ejemplo. class MyVector

El objetivo de esta aplicación es utilizar la clase MyVector, la cual permite representar secuencias de char en un vector privado, para leer una secuencia de char, invertir el orden y extraer los primeros 5 chars de esta secuencia invertida. El vector interno es estático y tiene memoria limitada, lo que va condicionar de forma importante la implementación.

### 9.1. myvector.h

```
#ifndef MYVECTOR_H
#define MYVECTOR_H
#include <string>
#define MAXVECTOR 10
#define ENDCHAR (char) '@'
#define EMPTYCHAR (char) '.'
#define NOCHAR (char) '\0'

class MyVector {
private:
    char letters[MAXVECTOR];
    int nletters;
public:
    int size() const { return nletters; };
    std::string to_string() const {
        std::string res="";
        for (int i=0; i<size();i++)
            res.push_back(letters[i]);
        return res;
    }
    std::string inspect() const {
        std::string res="";
        for (int i=0; i<MAXVECTOR;i++)
            res.push_back(letters[i]);
        return std::to_string(size())+" "+res;
    }
    //
    MyVector();
    void addLetter(char letter);
    void clear();
    char getLetter(int pos) const;
    void setLetter(int pos, char letter);
};
#endif /* MYVECTOR_H */
```

### 9.2. myvector.cpp

```
#include <string>
#include "myvector.h"
using namespace std;

MyVector::MyVector() { }

void MyVector::addLetter(char letter) { }

char MyVector::getLetter(int pos) const { }

void MyVector::setLetter(int pos, char letter) { }

void MyVector::clear() { }
```

### 9.3. main.cpp

```
#include <iostream>
#include "myvector.h"
#include "gtest/gtest.h"
#include "MPTests.h"
using namespace std;

int main() {

    // Step 1. Declare the needed variables
    MyVector initials, finals;

    // Step 2. Read the sequence from keyboard.

    // Step 3. Reverse the sequence without using any additional sequence

    // Step 4. Obtain a new sequence with the first 5 chars of the reversed sequence and
    // show it in the screen

}
```



## 10. Changelog

**V1** Contiene la práctica totalidad de características que hay descritas en este documento.

1. Extensión de Googletest
2. Permite reporte de tests inicial `./doc/markdown/ReadmeTests.md` mediante una script que parsea los ficheros de tests de la carpeta `./tests/`
3. Permite marcar la salida que se quiere validar `CVAL <<`
4. Permite trazar la evolución de ciertas clases a lo largo de la ejecución de main `REPORT_DATA(<instancia>)`
5. Permite saltar (ignorar) los tests siguientes cuando falla un test `SKIP_ASSERT`
6. Cubre tests de integración `DEF_EXECUTION_ENVIRONMENT(<label>)` con un mínimo impacto en el main de los alumnos, pues oculta la inicialización de Googletest `MAIN_BODY()` y el lanzamiento de los tests `RETURN_MAIN`
7. Simula lectura de datos desde teclado `FROM_KEYBOARD(<reading>)`
8. Simula redirección de entrada desde fichero `CALL_FROM_FILE()`
9. Permite simular el paso de parámetros a main `CALL(<arguments>)`
10. Permite dar un time-out para la ejecución y evitar que se cuelguen las aplicaciones `SET_TIMEOUT_S(<segundos>)`
11. Integra chequeo de memoria, tanto Valgrind como DR.Memory `USE_MEMORY_LEAKS(<tool>)`
12. Permite reportar el resultado de los tests en un documento auditable `./tests/TestReport.md`

**V2** A raíz de la corrección de la Práctica 0, surgen algunas necesidades para integrar mejor el desarrollo de los tests con la script de autocorrección, siempre con el objetivo de dar una salida más comprensible por los alumnos y totalmente homogénea tanto en la salida por pantalla, como en el reporte inicial de tests, como en el reporte de resultados y en el feedback que la script de autocorrección da a los alumnos.

1. Para poder implementar los cambios de esta versión, se ha tenido que integrar más profundamente con Googletest para poder definir un espacio de memoria compartido entre los tests y checkpoints de una misma suite de tests, de forma que lo que antes se expresaba como

```
TEST(<suite>, <test>)
```

ahora se expresa como

```
class <suite> : public ::testing::Test {
public:

<suite> : Test() {
}

protected:

virtual void SetUp() {
    <init shared variables>
}

virtual void TearDown() {
    <shutdown shared variables>
}

<declare shared variables>
};

TEST_F(<suite>, <test>)
```

Esta nueva organización no afecta al contenido de los tests, cuyo significado se mantiene el mismo que antes, pero habilita los principales cambios de esta versión. Dado que en la asignatura hemos creado tres suites diferentes según el nivel de complejidad: `_01_Basics`, `_02_Intermediate` y `_03_Advanced` las respectivas clases de test ya están declaradas en `MPTest.h` para mantener los ficheros de test con los mínimos cambios, es decir, únicamente se cambia `TEST()` por `TEST_F()`

2. Todos los tests y checkpoints están completa y biunívocamente referenciados con un índice doble `<#test>::<#checkpoint>` Este índice aparece en el informe inicial, en la pantalla al ejecutar `MPTest`, en el feedback al alumno tras la autocorrección y en el reporte de resultado de los tests, lo que permite identificar mejor al test que ha fallado para su revisión.
3. El reporte inicial, que ahora está indexado, se llama ahora

`./doc/markdown/TestDescription.md` y no se hace con una script, sino siguiendo el mismo proceso de testeo que se haría normalmente, lo que permite recorrer todos los tests y checkpoints *activos*. Este fichero con extensión md (markdown) se puede pasar directamente a LaTeX para su inclusión en los guiones mediante la orden (requiere el programa `pandoc`) `pandoc TestDescription.md -t latex -o TestDescription.tex` el cual se integra automáticamente como una sección independiente y autocontenida de los tests, a modo de manual de referencia para el desarrollador.

4. El reporte final ahora está indexado y contiene el mismo contenido que antes excepto en los tests de integración, en los que, para ser más asertivo, en caso de fallo de la salida del programa se cambia el escueto

`EXPECTED_OUTPUT must be equal to REAL_OUTPUT`

Por una salida integrada con la sintaxis del programa `wdiff` indicando qué partes de la salida faltan y qué partes sobran (es necesario tener instalado este programa). Ver secciones previas para conocer esta sintaxis de salida.

5. Soporte integrado para errores de segmentation fault. En caso de que se produzca un segmentation fault durante una prueba de integración, se ha mejorado la integración con `Valgrind` para detectar las posibles causas e informar de la línea en la que se produce el core. Por ejemplo, si en `Shopping1` no se implementa bien el método `int weekDay()`, cuyo resultado se utiliza como índice de un vector, se puede producir un error de acceso a memoria inesperado. En ese caso, basta con incluir el chequeo de `Valgrind` en el test y veremos como nos informa de que se ha producido un core, seguramente motivado por un acceso de lectura fuera de rango, e indica las líneas afectadas.

```
[ MEMCHECK ] ECommerce5-valgrind
[--BAD_READ] (main.cpp:106) (main.cpp:87)
[-CORE_DUMP] (main.cpp:106) (main.cpp:87)
```

Esta utilidad es MUY POTENTE de cara a `Shopping2` o, en general, para cualquier prueba de integración que implique el uso de vectores estáticos o dinámicos, porque permite detectar si los programas, llevados a tests extremos, se salen o no de los vectores, aunque ésto produzca un segmentation fault o, simplemente, no produzca ningún error evidente.

6. Para dotar a `REPORT_DATA(<objeto>)` de una información más contextual sobre el volcado de memoria del objeto, se ha definido la macro `REPORT_DATA2(<objeto>, mensaje)` de forma que se añada una etiqueta para indicar en qué parte del programa se está haciendo el volcado. Por ejemplo, este programa
-

```
int main() {
    Number a;
    REPORT_DATA(a);          /// Checkpoint inicial
    int increm1, increm2;
    cout << "Introduzca dos incrementos consecutivos ";
    cin >> increm1 >> increm2;
    for (int i=0; i<increm1; a.increase(), i++) REPORT_DATA(a);    /// Checkpoint
        continuo
    for (int i=0; i<increm2; a.increase(), i++);
    if (a.isEven()) {
        CVAL << "El resultado es par: " << a.get() << endl;
    } else {
        CVAL << "El resultado es impar: " << a.get() << endl;
    }
    REPORT_DATA(a);          /// Checkpoint final
}
```

Produce esta salida, en la que la sucesión de reportes puede ser algo confusa.

```
[a] {Number}::[0]
Introduzca dos incrementos consecutivos 4 1
[a] {Number}::[0]
[a] {Number}::[1]
[a] {Number}::[2]
[a] {Number}::[3]
El resultado es impar: 5
[a] {Number}::[5]
}
```






Ahora se puede usar esta nueva macro de manera más informativa.

```
int main() {
    Number a;
    REPORT_DATA2(a, "Iniciación"); /// @brief Checkpoint inicial
    int increm1, increm2, increm3;
    cout << "Introduzca dos incrementos consecutivos ";
    cin >> increm1 >> increm2;
    for (int i = 0; i < increm1; a.increase(), i++)
        REPORT_DATA2(a, (string)"Bucle for, iteración "+to_string(i)); ///
        @brief Checkpoint continuo
    for (int i = 0; i < increm2; a.increase(), i++);
    if (a.isEven()) {
        CVAL << "El resultado es par: " << a.get() << endl;
    } else {
        CVAL << "El resultado es impar: " << a.get() << endl;
    }
    REPORT_DATA2(a, "Valor final"); /// @brief Checkpoint final
}
```

permitiría obtener una salida más informativa. Introduzca dos incrementos consecutivos

```
[a(Iniciación)] {Number}::[0]
Introduzca dos incrementos consecutivos 4 1
[a(Bucle for, iteración 0)] {Number}::[0]
[a(Bucle for, iteración 1)] {Number}::[1]
[a(Bucle for, iteración 2)] {Number}::[2]
[a(Bucle for, iteración 3)] {Number}::[3]
El resultado es impar: 5
[a(Valor final)] {Number}::[5]
}
```

## 11. Videotutoriales

1. Introducción a TDD  (Abrir →)
2. Creando el proyecto  (Abrir →)
3. Primeros tests  (Abrir →)
4. Resultado de los tests  (Abrir →)
5. Tests de integración  (Abrir →)