Documentazione di GoogleTest v1.16.0 20251123 (Ita)

Release 1.16.0

GoogleTest Community

(Traduzione: Baldassarre Cesarano)

22 nov 2025

Contenuto:

CAPITOLO 1

Guida Utente di GoogleTest

1.1 Benvenuti in GoogleTest!

GoogleTest è il framework di test e mocking C++ di Google. Questa guida lutente ha i seguenti contenuti:

- Guida di GoogleTest - Insegna come scrivere semplici test utilizzando GoogleTest. I principianti dovrebbero cominciare da qui.
- GoogleTest Advanced - Leggere questo, dopo la Guida per utilizzare GoogleTest al massimo delle sue potenzialità.
- Esempi di Googletest - Descrive degli esempi di GoogleTest.
- FAQ di GoogleTest - Ci sono domande? Si cercano Suggerimenti? Controllare prima qui.
- Mocking per Principianti - Insegna come creare oggetti mock e come utilizzarli nei test.
- Mocking Cookbook - Include suggerimenti e approcci ai casi duso comuni di mocking.
- Cheat Sheet del Mocking - Un pratico riferimento per matcher, azioni, invarianti e altro.
- Mocking FAQ - Contiene le risposte ad alcune domande specifiche sul mocking.

CAPITOLO 2

How to become a contributor and submit your own code

2.1 Contributor License Agreements

Wed love to accept your patches! Before we can take them, we have to jump a couple of legal hurdles.

Please fill out either the individual or corporate Contributor License Agreement (CLA).

- If you are an individual writing original source code and youre sure you own the intellectual property, then youll need to sign an individual CLA.
- If you work for a company that wants to allow you to contribute your work, then youll need to sign a corporate CLA.

Follow either of the two links above to access the appropriate CLA and instructions for how to sign and return it. Once we receive it, well be able to accept your pull requests.

2.2 Are you a Googler?

If you are a Googler, please make an attempt to submit an internal contribution rather than a GitHub Pull Request. If you are not able to submit internally, a PR is acceptable as an alternative.

2.3 Contributing A Patch

1. Submit an issue describing your proposed change to the issue tracker.
2. Please dont mix more than one logical change per submittal, because it makes the history hard to follow. If you want to make a change that doesnt have a corresponding issue in the issue tracker, please create one.
3. Also, coordinate with team members that are listed on the issue in question. This ensures that work isnt being duplicated and communicating your plan early also generally leads to better patches.
4. If your proposed change is accepted, and you havent already done so, sign a Contributor License Agreement (*see details above*).
5. Fork the desired repo, develop and test your code changes.
6. Ensure that your code adheres to the existing style in the sample to which you are contributing.
7. Ensure that your code has an appropriate set of unit tests which all pass.
8. Submit a pull request.

2.4 The Google Test and Google Mock Communities

The Google Test community exists primarily through the discussion group and the GitHub repository. Likewise, the Google Mock community exists primarily through their own discussion group. You are definitely encouraged to contribute to the discussion and you can also help us to keep the effectiveness of the group high by following and promoting the guidelines listed here.

2.4.1 Please Be Friendly

Showing courtesy and respect to others is a vital part of the Google culture, and we strongly encourage everyone participating in Google Test development to join us in accepting nothing less. Of course, being courteous is not the same as failing to constructively disagree with each other, but it does mean that we should be respectful of each other when enumerating the 42 technical reasons that a particular proposal may not be the best choice. Theres never a reason to be antagonistic or dismissive toward anyone who is sincerely trying to contribute to a discussion.

Sure, C++ testing is serious business and all that, but its also a lot of fun. Lets keep it that way. Lets strive to be one of the friendliest communities in all of open source.

As always, discuss Google Test in the official GoogleTest discussion group. You dont have to actually submit code in order to sign up. Your participation itself is a valuable contribution.

2.5 Style

To keep the source consistent, readable, diffable and easy to merge, we use a fairly rigid coding style, as defined by the google-styleguide project. All patches will be expected to conform to the style outlined here. Use .clang-format to check your formatting.

2.6 Requirements for Contributors

If you plan to contribute a patch, you need to build Google Test, Google Mock, and their own tests from a git checkout, which has further requirements:

- Python v3.6 or newer (for running some of the tests and re-generating certain source files from templates)
- CMake v2.8.12 or newer

2.7 Developing Google Test and Google Mock

This section discusses how to make your own changes to the Google Test project.

2.7.1 Testing Google Test and Google Mock Themselves

To make sure your changes work as intended and dont break existing functionality, youll want to compile and run Google Test and GoogleMocks own tests. For that you can use CMake:

```
mkdir mybuild
cd mybuild
cmake -Dgtest_build_tests=ON -Dgmock_build_tests=ON ${GTEST_REPO_DIR}
```

To choose between building only Google Test or Google Mock, you may modify your cmake command to be one of each

```
cmake -Dgtest_build_tests=ON ${GTEST_DIR} # sets up Google Test tests
cmake -Dgmock_build_tests=ON ${GMOCK_DIR} # sets up Google Mock tests
```

Make sure you have Python installed, as some of Google Tests tests are written in Python. If the cmake command complains about not being able to find Python (`Could NOT find PythonInterp (missing: PYTHON_EXECUTABLE)`), try telling it explicitly where your Python executable can be found:

```
cmake -DPYTHON_EXECUTABLE=path/to/python ...
```

Next, you can build Google Test and / or Google Mock and all desired tests. On *nix, this is usually done by

```
make
```

To run the tests, do

```
make test
```

All tests should pass.

CAPITOLO 3

GoogleTest

3.1 Announcements

3.1.1 Documentation Updates

Our documentation is now live on GitHub Pages at https://google.github.io/googletest/. We recommend browsing the documentation on GitHub Pages rather than directly in the repository.

3.1.2 Release 1.17.0

Release 1.17.0 is now available.

The 1.17.x branch requires at least C++17.

3.1.3 Continuous Integration

We use Googles internal systems for continuous integration.

3.1.4 Coming Soon

- We are planning to take a dependency on Abseil.

3.2 Welcome to GoogleTest, Googles 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.

3.2.1 Iniziamo

See the GoogleTest Users Guide for documentation. We recommend starting with the GoogleTest Primer.

More information about building GoogleTest can be found at *googletest/README.md*.

3.3 Features

- xUnit test framework:
  Googletest is based on the xUnit testing framework, a popular architecture for unit testing
- Test discovery:
  Googletest automatically discovers and runs your tests, eliminating the need to manually register your tests
- Rich set of assertions:
  Googletest provides a variety of assertions, such as equality, inequality, exceptions, and more, making it easy to test your code
- User-defined assertions:
  You can define your own assertions with Googletest, making it simple to write tests that are specific to your code
- Death tests:
  Googletest supports death tests, which verify that your code exits in a certain way, making it useful for testing error-handling code
- Fatal and non-fatal failures:
  You can specify whether a test failure should be treated as fatal or non-fatal with Googletest, allowing tests to continue running even if a failure occurs
- Value-parameterized tests:
  Googletest supports value-parameterized tests, which run multiple times with different input values, making it useful for testing functions that take different inputs
- Type-parameterized tests:
  Googletest also supports type-parameterized tests, which run with different data types, making it useful for testing functions that work with different data types
- Various options for running tests:
  Googletest provides many options for running tests including running individual tests, running tests in a specific order and running tests in parallel

3.4 Piattaforme Supportate

GoogleTest segue le Foundational C++ Support Policy di Google. See this table for a list of currently supported versions of compilers, platforms, and build tools.

3.5 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, Googles data interchange format.
- The OpenCV computer vision library.

3.6 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. GoogleTest 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.

3.7 Contributing Changes

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

Happy testing!

CAPITOLO 4

Piattaforme Supportate

GoogleTest segue le Foundational C++ Support Policy di Google. Consultare questa tabella per un elenco delle versioni di compilatori, di piattaforme e di strumenti di building attualmente supportati.

CAPITOLO 5

Quickstart: Il Building con Bazel

Questo tutorial ha lo scopo rendere operativi con luso di GoogleTest col sistema di build Bazel. Se è la prima volta che si usa GoogleTest, o cè bisogno di un ripasso, consigliamo questo tutorial come punto di partenza.

5.1 Prerequisiti

Per questo tutorial, cè bisogno di:

- Un sistema operativo compatibile (ad esempio Linux, macOS, Windows).
- A compatible C++ compiler that supports at least C++17.
- Bazel 7.0 or higher, the preferred build system used by the GoogleTest team.

Vedere Supported Platforms per ulteriori informazioni sulle piattaforme compatibili con GoogleTest.

Se Bazel non è ancora installato, consultare la Bazel installation guide.

{: .callout .note} Nota: I comandi del terminale in questo tutorial mostrano un prompt della shell Unix, ma funzionano anche sulla riga di comando di Windows.

5.2 Configurare un workspace Bazel

Un workspace Bazel è una directory sul file system che si utilizza per gestire i file sorgenti per il software che si desideri creare. Each workspace directory has a text file named `MODULE.bazel` which may be empty, or may contain references to external dependencies required to build the outputs.

Innanzitutto, si crea una directory per il workspace:

```
$ mkdir my_workspace && cd my_workspace
```

Next, youll create the `MODULE.bazel` file to specify dependencies. As of Bazel 7.0, the recommended way to consume GoogleTest is through the Bazel Central Registry. To do this, create a `MODULE.bazel` file in the root directory of your Bazel workspace with the following content:

```
# MODULE.bazel

# Choose the most recent version available at
# https://registry.bazel.build/modules/googletest
bazel_dep(name = "googletest", version = "1.17.0")
```

Ora si è pronti per creare codice C++ che utilizza GoogleTest.

5.3 Crea ed eseguire un file binario

Una volta configurato il workspace Bazel, si può utilizzare il codice GoogleTest allinterno del progetto.

Come esempio, si crea un file denominato `hello_test.cc` nella directory `my_workspace` col seguente contenuto:

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

// Demonstrate some basic assertions.
TEST(HelloTest, BasicAssertions) {
  // Expect two strings not to be equal.
  EXPECT_STRNE("hello", "world");
  // Expect equality.
  EXPECT_EQ(7 * 6, 42);
}
```

GoogleTest fornisce asserzioni utilizzate per testare il comportamento del codice. Lesempio precedente include il file header principale di GoogleTest e mostra alcune asserzioni di base.

Per buildare il codice, si crea un file denominato `BUILD` nella stessa directory con il seguente contenuto:

```
cc_test(
    name = "hello_test",
    size = "small",
    srcs = ["hello_test.cc"],
    deps = [
        "@googletest//:gtest",
        "@googletest//:gtest_main",
    ],
)
```

This `cc_test` rule declares the C++ test binary you want to build, and links to the GoogleTest library (`@googletest//:gtest`") and the GoogleTest `main()` function (`@googletest//:gtest_main`). Per ulteriori informazioni sui file Bazel `BUILD`, vedere il Bazel C++ Tutorial.

{: .callout .note} NOTE: In the example below, we assume Clang or GCC and set `--cxxopt=-std=c++17` to ensure that GoogleTest is compiled as C++17 instead of the compilers default setting. For MSVC, the equivalent would be `--cxxopt=/std:c++17`. Vedere Supported Platforms per ulteriori dettagli sulle versioni dei linguaggi supportate.

Ora si possono buildare ed eseguire il test:

Congratulazioni! È stato creato ed eseguito correttamente un file binario di prova utilizzando GoogleTest.

5.4 Passi successivi

- Consultare la Guida per iniziare a scrivere semplici test.
- Esaminare gli esempi di codice per altri esempi che mostrano come utilizzare le varie funzioni di GoogleTest.

CAPITOLO 6

Quickstart: Il Building con CMake

Questo tutorial ha lo scopo rendere operativi con luso di GoogleTest con CMake. Se è la prima volta che si usa GoogleTest, o cè bisogno di un ripasso, consigliamo questo tutorial come punto di partenza. Se il progetto utilizza Bazel, consultare invece Quickstart for Bazel.

6.1 Prerequisiti

Per questo tutorial, cè bisogno di:

- Un sistema operativo compatibile (ad esempio Linux, macOS, Windows).
- A compatible C++ compiler that supports at least C++17.
- CMake e uno strumento di build compatibile per il progetto.
  - Gli strumenti di build compatibili includono Make, Ninja ed altri - consultare CMake Generators per ulteriori informazioni.

Vedere Supported Platforms per ulteriori informazioni sulle piattaforme compatibili con GoogleTest.

Se Bazel non è ancora installato, consultare la CMake installation guide.

{: .callout .note} Nota: I comandi del terminale in questo tutorial mostrano un prompt della shell Unix, ma funzionano anche sulla riga di comando di Windows.

6.2 Preparazione di un progetto

CMake usa un file chiamato `CMakeLists.txt` per configurare il sistema di build per un progetto. Si utilizzerà questo file per impostare il progetto e per dichiarare una dipendenza da GoogleTest.

Innanzitutto, si crea una directory per il progetto:

```
$ mkdir my_project && cd my_project
```

Poi, si creerà il file `CMakeLists.txt` e si dichiarerà una dipendenza da GoogleTest. Esistono molti modi per esprimere le dipendenze nellecosistema CMake; in questa guida rapida si userà `FetchContent` di CMake](https://cmake.org/cmake/help/latest/module/FetchContent.html). Per farlo, nella directory del progetto (`my_project`), si crea un file chiamato `CMakeLists.txt` con i seguenti contenuti:

```
cmake_minimum_required(VERSION 3.14)
project(my_project)

# GoogleTest requires at least C++17
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

include(FetchContent)
FetchContent_Declare(
  googletest
  URL https://github.com/google/googletest/archive/
↪03597a01ee50ed33e9dfd640b249b4be3799d395.zip
)
# For Windows: Prevent overriding the parent project's compiler/linker settings
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)
```

La configurazione precedente dichiara una dipendenza da GoogleTest che viene scaricato da GitHub. Nellesempio precedente, `03597a01ee50ed33e9dfd640b249b4be3799d395` è lhash del commit Git della versione di GoogleTest da utilizzare; consigliamo di aggiornare spesso lhash in modo che punti alla versione più recente.

Per ulteriori informazioni su come creare i file `CMakeLists.txt`, consultare il Tutorial su CMake.

6.3 Crea ed eseguire un file binario

Con GoogleTest dichiarato come dipendenza, se ne può utilizzare il codice allinterno del progetto.

Per esempio, creare un file chiamato `hello_test.cc` nella directory `my_project` col seguente contenuto:

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

// Demonstrate some basic assertions.
TEST(HelloTest, BasicAssertions) {
  // Expect two strings not to be equal.
  EXPECT_STRNE("hello", "world");
  // Expect equality.
  EXPECT_EQ(7 * 6, 42);
}
```

GoogleTest fornisce asserzioni utilizzate per testare il comportamento del codice. Lesempio precedente include il file header principale di GoogleTest e mostra alcune asserzioni di base.

Per creare il codice, si aggiunge quanto segue alla fine del file `CMakeLists.txt`:

```
enable_testing()

add_executable(
  hello_test
  hello_test.cc
)
target_link_libraries(
  hello_test
  GTest::gtest_main
)

include(GoogleTest)
gtest_discover_tests(hello_test)
```

La configurazione precedente abilita il test in CMake, dichiara il file binario di test C++ che si desidera buildare (`hello_test`) e lo ńlinkaż a GoogleTest (`gtest_main`). Le ultime due righe consentono al runner dei test di CMake di scoprire i test inclusi nel file binario, utilizzando il modulo `GoogleTest` di CMake.

Ora si possono buildare ed eseguire il test:

Congratulazioni! È stato creato ed eseguito correttamente un file binario di prova utilizzando GoogleTest.

6.4 Passi successivi

- Consultare la Guida per iniziare a scrivere semplici test.
- Esaminare gli esempi di codice per altri esempi che mostrano come utilizzare le varie funzioni di GoogleTest.

CAPITOLO 7

Guida di GoogleTest

7.1 Introduzione: Perché GoogleTest?

GoogleTest aiuta a scrivere meglio i test di C++.

GoogleTest è un framework per i test sviluppato dal team ńTesting Technologyż tenendo presenti i requisiti e i vincoli specifici di Google. Sia se si lavora su Linux, su Windows, o su Mac, se si scrive codice C++, GoogleTest può essere utile. Supporta *qualsiasi* tipo di test, non solo le ńunit testż.

Quindi cosa rende buon un test e come si inserisce GoogleTest? Noi crediamo che:

1. I test dovrebbero essere *independenti* e *ripetibili*. È seccante eseguire il debug di un test che ha esito positivo o negativo a seconda di altri test. GoogleTest isola i test eseguendo ciascuno di essi su un oggetto diverso. Quando un test fallisce, GoogleTest consente di eseguirlo in isolamento per un debug rapido.
2. I test dovrebbero essere ben *organizzati* e riflettere la struttura del codice testato. GoogleTest raggruppa i test correlati in suite di test che possono condividere dati e subroutine. Questo schema comune è facile da riconoscere e semplifica la manutenzione dei test. Tale coerenza è particolarmente utile quando si cambia progetto e si inizia a lavorare su una nuova base di codice.
3. I test dovrebbero essere *portabili* e *riutilizzabili*. Google possieme molto codice indipendente dalla piattaforma; anche i suoi test dovrebbero anche essere indipendenti dalla piattaforma. GoogleTest funziona su diversi sistemi operativi, con vari compilatori, con o senza eccezioni, quindi i test di GoogleTest possono funzionare con una varietà di configurazioni.
4. Quando i test falliscono, dovrebbero fornire quante più *informazioni* possibili sul problema. GoogleTest non si ferma al primo fallimento del test. Ma, interrompe solo il test corrente e continua col successivo. È inoltre possibile impostare test che riportino ad errori non fatali dopo i quali il test corrente continua. Pertanto, si possono rilevare e correggere più bug in un singolo ciclo di esecuzione-modifica-compilazione.
5. Il framework di testing dovrebbe liberare gli autori dalle ńfaccende domestichež e consentire loro di concentrarsi sul *contenuto* del test. GoogleTest tiene traccia automaticamente di tutti i test definiti e non richiede allutente di elencarli per eseguirli.
6. I test dovrebbero essere *veloci*. Con GoogleTest si possono le risorse condivise tra i test e impegnarsi per linstallazione/de-installazione una sola volta, senza che i test dipendano gli uni dagli altri.

Dat che GoogleTest è basato sulla popolare architettura xUnit, ci si sentirà come a casa se si è già utenti di JUnit o di PyUnit. In caso contrario, ci vorranno circa 10 minuti per apprendere le nozioni di base e cominciare. Quindi andiamo!

7.2 Attenzione alla Nomenclatura

{: .callout .note} *Nota:* Potrebbe verificarsi un po di confusione derivante dalle diverse definizioni dei termini *Test*, *Test Case* e *Test Suite*, quindi attenzione a non confonderli.

Storicamente, GoogleTest ha iniziato a utilizzare il termine *Test Case* per raggruppare test correlati, mentre le pubblicazioni attuali, inclusi i materiali dellInternational Software Testing Qualifications Board (ISTQB) e vari libri di testo sulla qualità del software, utilizzano il termine *[Test Suite][istqb test suite]* per questo.

Il termine correlato *Test*, così come viene utilizzato in GoogleTest, corrisponde al termine *[Test Case][istqb test case]* di ISTQB e di altri.

Il termine *Test* ha comunemente un senso sufficientemente ampio, inclusa la definizione di *Test Case*, di ISTQB, quindi non è un grosso problema. Ma il termine *Test Case* così come è stato utilizzato in Google Test ha un senso contraddittorio e quindi crea confusione.

GoogleTest ha recentemente iniziato a sostituire il termine *Test Case* con *Test Suite*. LAPI preferita è *TestSuite*. La vecchia API TestCase viene lentamente deprecata e sottoposta a refactoring.

Si prega quindi di prestare attenzione alle diverse definizioni dei termini:

Significato	Termine GoogleTest	Termine ISTQB
Testare un particolare path del programma con valori di input specifici e verificare i risultati	*TEST()*	Test Case

7.3 Concetti Base

Quando si usa GoogleTest, si comincia scrivendo *asserzioni*, ovvero affermazioni che verificano se una condizione è vera. Il risultato di unasserzione può essere *success*, *nonfatal failure* o *fatal failure*. Se avviane una ńfatal failureż, interrompe la funzione corrente; altrimenti il programma continua normalmente.

Tests utilizzano le asserzioni per verificare il comportamento del codice in esame. Se un test va in crash o ha fallisce unasserzione, allora *fail*; altrimenti *succeeds*.

Una *test suite* contiene uno o più test. I test si dovrebbero raggruppare in test suite che riflettono la struttura del codice in esame. Quando più test, in una test suite, devono condividere oggetti e subroutine comuni, si possono inserire in una classe *test fixture*.

Un *test program* può contenere più test suite.

Spiegheremo ora come scrivere un ńtest programż, partendo dal livello della singola asserzione e costruendo poi ńtestż e ńtest suiteż.

7.4 Le Asserzioni

Le asserzioni di GoogleTest sono macro che assomigliano alle chiamate alle funzioni. Una classe o una funzione si può testare facendo asserzioni sul suo comportamento. Quando unasserzione fallisce, GoogleTest stampa il file sorgente dellasserzione e la posizione del numero di riga, insieme a un messaggio di errore. Si può anche avere un messaggio di errore personalizzato che verrà aggiunto al messaggio di GoogleTest.

Le asserzioni si presentano in coppie che mettono alla prova la stessa cosa ma hanno effetti diversi sulla funzione corrente. Le versioni `ASSERT_*` generano errori fatali quando falliscono e **interrompono la funzione corrente**. Le versioni `EXPECT_*` generano errori non fatali, che non interrompono la funzione corrente. Solitamente si preferiscono le `EXPECT_*`, poiché consentono di segnalare più di un errore in un test. Tuttavia, si usa `ASSERT_*` se non ha senso continuare quando lasserzione in questione fallisce.

Poiché un `ASSERT_*` non riuscito ritorna immediatamente dalla funzione corrente, probabilmente saltando il codice di pulizia successivo, potrebbe causare dei ńmemory leakż. A seconda della natura del leak, potrebbe valere o meno la pena risolverlo, quindi lo si tenga a mente se si riceve un errore sul controllo dellheap oltre agli errori dellasserzione.

Per avere un messaggio di errore personalizzato, è sufficiente inserirlo nella macro utilizzando loperatore « o una sequenza di tali operatori. Esaminare lesempio seguente, utilizzando le macro *ASSERT_EQ e EXPECT_EQ* per verificare luguaglianza dei valori:

```
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;
}
```

Tutto ciò che può essere inviato a un `ostream` e può essere inviato ad una macro di asserzione–in particolare, stringhe C e oggetti `string`. Se una stringa wide (`wchar_t*`, `TCHAR*` in modalità `UNICODE` su Windows, o `std::wstring`) viene inviata in streaming a unasserzione, verrà tradotto in UTF-8 una volta stampato.

GoogleTest fornisce una raccolta di asserzioni per verificare il comportamento del codice in vari modi. Si possono controllare le condizioni booleane, confrontare valori in base a operatori relazionali, verificare valori stringa, valori in virgola mobile e molto altro. Esistono anche asserzioni che consentono di verificare stati più complessi fornendo predicati personalizzati. Per lelenco completo delle asserzioni fornite da GoogleTest, consultare i *Riferimenti sulle Asserzioni*.

7.5 Test Semplici

Per creare un test:

1. Si usa la macro `TEST()` per definire e dare un nome a una funzione di test. Queste sono normali funzioni C++ che non restituiscono un valore.
2. In questa funzione, insieme a qualsiasi istruzione C++ valida che si voglia includere, si utilizzano le varie asserzioni di GoogleTest per verificare i valori.
3. Il risultato del test è determinato dalle asserzioni; se una qualsiasi asserzione nel test fallisce (in modo fatale o non fatale), o se il test fallisce, lintero test fallisce. Altrimenti, è riuscito.

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

Gli argomenti di `TEST()` vanno dal generale allo specifico. Il *primo* argomento è il nome della ńtest suiteż e il *secondo* argomento è il nome del test nella ńtest suiteż. Entrambi i nomi devono essere identificatori C++ validi e non devono contenere underscore (_). Il *nome completo* di un test è costituito dalla test suite che lo contiene e dal suo nome individuale. I test di diverse test suite possono avere lo stesso nome individuale.

Ad esempio, prendiamo una semplice funzione intera:

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

Una test suite per questa funzione potrebbe essere simile a:

```
// 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 raggruppa i risultati dei test per ńtest suiteż, quindi i test logicamente correlati dovrebbero trovarsi nella stessa test suite; in altre parole, il primo argomento del loro `TEST()` dovrebbe essere lo stesso. Nellesempio precedente, abbiamo due test, `HandlesZeroInput` e `HandlesPositiveInput`, che appartengono alla test suite `FactorialTest`.

Quando si dà un nome alle test suite e ai test, si deve seguire la stessa convenzione usata per denominare funzioni e classi.

Disponibilità: Linux, Windows, Mac.

7.6 Test Fixture: Uso della Stessa Configurazione dei Dati per Più Test {#same-data-multiple-tests}

Se ci si ritrova a scrivere due o più test che operano su dati simili, si può usare una *test fixture*. Questa consente di riutilizzare la stessa configurazione di oggetti per più test.

Per creare una fixture:

1. Si deriva una classe da `testing::Test` . Il body inizia con `protected:`, per accedere ai membri della fixture dalle sottoclassi.
2. Allinterno della classe, si dichiarano tutti gli oggetti che si intendono usare.
3. Se necessario, si scrive un costruttore di default o una funzione `SetUp()` per preparare gli oggetti per ciascun test. Un errore comune è scrivere `SetUp()` come **`Setup()`** con una u minuscola - Usare `override` in C++11 per essere sicuri di scriverlo correttamente.
4. Se necessario, scrivere un distruttore o una funzione `TearDown()` per rilasciare eventuali risorse allocate in `SetUp()`. Per sapere quando usare il costruttore/distruttore e quando usare `SetUp()/TearDown()`, leggere le FAQ.
5. Se necessario, definire delle subroutine da condividere tra i test.

Quando si usa una fixture, si usa `TEST_F()` anziché `TEST()` in quanto consente di accedere a oggetti e subroutine nella fixture di test:

```
TEST_F(TestFixtureClassName, TestName) {
  ... test body ...
}
```

A differenza di `TEST()`, in `TEST_F()` il primo argomento deve essere il nome della classe della fixture di test. (`_F` sta per ńFixtureż). Per questa macro non è specificato alcun nome di test suite.

Sfortunatamente, il sistema delle macro di C++ non ci consente di creare ununica macro per gestire entrambi i tipi di test. Lutilizzo della macro errata provoca un errore del compilatore.

Inoltre, si deve definire una classe per la fixture di test prima di usarla in una `TEST_F()`, altrimenti si riceve lerrore del compilatore ńvirtual outside class declarationż.

Per ogni test definito con `TEST_F()`, GoogleTest creerà una *nuova* fixture a runtime, lo inizializzerà immediatamente tramite `SetUp()`, esegue il test, ripulisce chiamando `TearDown()` ed esegue il delete della fixture di test. Si tenga presente che test diversi nella stessa suite di test hanno oggetti fixture di test diversi e GoogleTest elimina sempre una fixture di test prima di creare quella successiva. GoogleTest **non** riutilizza la stessa fixture per più test. Qualsiasi modifica apportata da un test alla fixture non influisce sugli altri test.

Ad esempio, scriviamo i test per una classe di una coda FIFO chiamata `Queue`, che ha la seguente interfaccia:

```
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.
```

(continues on next page)

(continua dalla pagina precedente)

```
  size_t size() const;
  ...
};
```

Innanzitutto, si definisce una classe fixture. Per convenzione, si da il nome `FooTest` dove `Foo` è la classe da testare.

```
class QueueTest : public testing::Test {
 protected:
  QueueTest() {
     // q0_ remains empty
     q1_.Enqueue(1);
     q2_.Enqueue(2);
     q2_.Enqueue(3);
  }

  // ~QueueTest() override = default;

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

In this case, we dont need to define a destructor or a `TearDown()` method, because the implicit destructor generated by the compiler will perform all of the necessary cleanup.

Ora scriveremo i test utilizzando `TEST_F()` e questa 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;
}
```

Quanto sopra utilizza entrambe le asserzioni `ASSERT_*` e `EXPECT_*`. La regola pratica è quella di utilizzare `EXPECT_*` quando si desidera che il test continui a scoprire altri errori dopo il fallimento dellasserzione e utilizzare `ASSERT_*` quando continuare dopo il fallimento non ha senso. Ad esempio, la seconda asserzione nel test `Dequeue` è `ASSERT_NE(n, nullptr)`, poiché in seguito dobbiamo dereferenziare il puntatore `n`, che porterebbe a un segfault quando `n` è `NULL`.

Quando vengono eseguiti questi test, accade quanto segue:

1. GoogleTest costruisce un oggetto `QueueTest` (chiamiamolo `t1`).
2. Il primo test (`IsEmptyInitially`) viene eseguito su `t1`.

3. `t1` viene distrutto.
4. I passaggi precedenti vengono ripetuti su un altro oggetto `QueueTest`, questa volta eseguendo il test `DequeueWorks`.

Disponibilità: Linux, Windows, Mac.

7.7 Invocare i Test

`TEST()` e `TEST_F()` registrano implicitamente i loro test con GoogleTest. Pertanto, a differenza di molti altri framework di test C++, non è necessario elencare nuovamente tutti i test definiti per eseguirli.

Dopo aver definito tuoi test, si possono eseguire con `RUN_ALL_TESTS()`, che restituisce `0` se tutti i test hanno esito positivo, o `1` in caso contrario. Notare che `RUN_ALL_TESTS()` esegue *tutti i test* nella [link unit]–possono appartenere a diverse test suite, o anche a diversi file sorgenti.

Quando viene richiamata, la macro `RUN_ALL_TESTS()`:

- Salva lo stato di tutti i flag di GoogleTest.
- Crea un oggetto fixture per il primo test.
- Lo inizializza tramite `SetUp()`.
- Esegue il test sulloggetto fixture.
- Ripulisce la fixture tramite `TearDown()`.
- Cancella la fixture.
- Ripristina lo stato di tutti i flag di GoogleTest.
- Ripete i passaggi precedenti per il test successivo, fino allesecuzione di tutti i test.

Se si verifica un errore irreversibile, i passaggi successivi verranno saltati.

{: .callout .important}

> IMPORTANTE: **Non** si deve ignorare il valore di ritorno di `RUN_ALL_TESTS()` altrimenti si riceve un errore del compilatore. La logica di questa progettazione è che il servizio di test automatizzato determina se un test è stato superato in base al suo codice di uscita, non alloutput stdout/stderr; quindi la funzione `main()` deve restituire il valore di `RUN_ALL_TESTS()`.
>
> Inoltre, si deve chiamare `RUN_ALL_TESTS()` solo **una volta**. Chiamandola più volte, si entra in conflitto con delle funzionalità avanzate di GoogleTest (ad esempio, i death tests) e quindi non è supportato.

Disponibilità: Linux, Windows, Mac.

7.8 Scrivere la Funzione main()

La maggior parte degli utenti *non* dovrebbe aver bisogno di scrivere la propria funzione `main` e linkare invece `gtest_main` (al contrario di `gtest`), che definisce un punto di ingresso adeguato. Per i dettagli vedere la fine di questa sezione. Il resto di questa sezione dovrebbe applicarsi solo quando è necessario fare qualcosa di personalizzato prima dellesecuzione dei test che non possa essere espresso nellambito delle fixture e delle test suite.

Se si scrive la propria funzione `main`, essa deve restituire il valore di `RUN_ALL_TESTS()`.

Si può iniziare da questo prototipo:

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

#include <gtest/gtest.h>

namespace my {
```

(continues on next page)

(continua dalla pagina precedente)

```
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();
}
```

La funzione `testing::InitGoogleTest()` analizza la riga di comando per i flag di GoogleTest e rimuove tutti i

flag riconosciuti. Ciò consente allutente di controllare il comportamento di un programma di test tramite vari flag, di cui parleremo nella AdvancedGuide. Si **deve** chiamare questa funzione prima di chiamare `RUN_ALL_TESTS()`, altrimenti i flag non verranno inizializzati correttamente.

Su Windows, `InitGoogleTest()` funziona anche con stringhe estese [wide], quindi può essere utilizzato anche in programmi compilati in modalità `UNICODE`.

Ma forse si sta pensando che scrivere tutte queste funzioni di `main` sia troppo? Siamo completamente daccordo ed è per questo che Google Test fornisce unimplementazione di base di main(). Se soddisfa le esigenze, basta linkare il test alla libreria `gtest_main` e si è a posto.

{: .callout .note} NOTA: `ParseGUnitFlags()` è deprecato a favore di `InitGoogleTest()`.

7.9 Limitazioni Note

- Google Test è progettato per essere thread-safe. Limplementazione è thread-safe sui sistemi in cui è disponibile la libreria `pthreads`. Al momento è *unsafe* utilizzare le asserzioni di Google Test da due thread contemporaneamente su altri sistemi (ad esempio Windows). Nella maggior parte dei test questo non è un problema poiché di solito le asserzioni vengono fatte nel thread principale. Se si vuole aiutare, ci si può offrire volontari per implementare le primitive di sincronizzazione necessarie in `gtest-port.h` per la propria piattaforma.

CAPITOLO 8

Argomenti avanzati di GoogleTest

8.1 Introduzione

Dopo aver letto la Guida di GoogleTest e imparato a scrivere test utilizzando GoogleTest, è il momento di imparare dei nuovi trucchi. Questo documento illustrerà altre asserzioni e come costruire messaggi di errore complessi, propagare errori fatali, riutilizzare e velocizzare fixture e utilizzare vari flag con i test.

8.2 Altre Asserzioni

Questa sezione copre alcune asserzioni usate meno frequentemente, ma comunque significative.

8.2.1 Successi e Fallimento Espliciti

Vedere *Explicit Success and Failure* nei ńRiferimenti sulle Asserzioniż.

8.2.2 Asserzioni di Eccezioni

Vedere *Exception Assertions* nelle ńAsserzioni di Eccezioniż.

8.2.3 Asserzioni di Predicati per del Migliori Messaggi di Errore

Anche se GoogleTest dispone di un ricco insieme di asserzioni, queste non potranno mai essere complete, poiché è impossibile (e non è nemmeno una buona idea) prevedere tutti gli scenari in cui un utente potrebbe imbattersi. Pertanto, a volte un utente deve utilizzare `EXPECT_TRUE()` per verificare unespressione complessa, in mancanza di una macro migliore. Questo ha il problema di non mostrare i valori delle parti dellespressione, rendendo difficile capire cosa è andato storto. Come soluzione alternativa, alcuni utenti scelgono di costruirsi da soli il messaggio di errore, accodandolo a `EXPECT_TRUE()`. Tuttavia, ciò risulta problematico, soprattutto quando lespressione presenta effetti collaterali o è costosa da valutare.

GoogleTest offre tre diverse opzioni per risolvere questo problema:

Utilizzo di una Funzione Booleana Esistente

Se cè già una funzione o un funtore che restituisce `bool` (o un tipo che può essere convertito implicitamente in `bool`), lo si può usare in un *predicate assertion* (asserzione di predicato) per ottenere gratuitamente la stampa degli argomenti della funzione. Vedere *`EXPECT_PRED*`* nei ńRiferimenti sulle Asserzioniż per i dettagli.

Utilizzo di una Funzione che Restituisce un AssertionResult

Sebbene `EXPECT_PRED*()` e simili siano utili per un lavoro veloce, la sintassi non è soddisfacente: si devono usare macro diverse per caratteristiche diverse, e sembra più Lisp che C++. La classe `::testing::AssertionResult` risolve questo problema.

Un oggetto `AssertionResult` rappresenta il risultato di unasserzione (sia che si tratti di un successo che di un fallimento, e un messaggio associato). Un `AssertionResult` si può creare utilizzando una di queste funzioni ńfactoryż:

```
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();

}
```

È poi possibile utilizzare loperatore « per lo streaming dei messaggi alloggetto `AssertionResult`.

Per avere messaggi più leggibili nelle asserzioni booleane (ad esempio `EXPECT_TRUE()`), si scrive una funzione predicato che restituisca `AssertionResult` anziché `bool`. Per esempio, se si definisce `IsEven()` come:

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

invece di:

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

lasserzione fallita, `EXPECT_TRUE(IsEven(Fib(4)))`, stamperà:

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

invece di un più opaco

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

Per ottenere messaggi informativi anche in `EXPECT_FALSE` e in `ASSERT_FALSE` (un terzo delle asserzioni booleane nel codebase di Google sono negative) e si è daccordo nel rendere il predicato più lento in il caso di successo, si può fornire un messaggio in caso di successo:

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

(continues on next page)

(continua dalla pagina precedente)

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

Quindi verrà stampata listruzione `EXPECT_FALSE(IsEven(Fib(6)))`

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

Utilizzo di un Predicato-Formatter

Se si ritiene insoddisfacente il messaggio di default generato da *EXPECT_PRED** e da *EXPECT_TRUE* o alcuni argomenti del predicato supportano lo streaming su `ostream`, si possono alternativamente utilizzare *predicate-formatter assertions* per personalizzare *completamente* la modalità di formattazione del messaggio. Vedere *EXPECT_PRED_FORMAT** nei ńRiferimenti sulle Asserzioniż per i dettagli.

8.2.4 Confronto in Virgola Mobile

Vedere *Floating-Point Comparison* nei ńRiferimenti sulle Asserzioniż.

Predicato in Virgola Mobile-Funzioni di Formattazione

Alcune operazioni in virgola mobile sono utili, ma non vengono utilizzate tanto spesso. Per evitare unesplosione di nuove macro, le forniamo come funzioni in formato predicato, utilizzabili nella macro di asserzione del predicato *EXPECT_PRED_FORMAT2*, ad esempio:

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

Il codice precedente controlla che `val1` sia inferiore o approssimativamente uguale a `val2`.

8.2.5 Asserzioni con luso dei Matcher di gMock

Vedere *EXPECT_THAT* nelle ńAsserzioni di Eccezioniż.

8.2.6 Altre Asserzioni di Stringa

(Se non è stato fatto, è opportuno leggere prima la sezione *precedente*).

Si possono usare i *matcher di stringhe* di gMock con *EXPECT_THAT* per fare altri tipi di confronti tra stringhe (sotto-stringa, prefisso, suffisso, espressione regolare, ecc.). Per esempio,

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

8.2.7 Asserzioni HRESULT di Windows

Vedere *Windows HRESULT Assertions* nelle ńAsserzioni di Eccezioniż.

8.2.8 Tipi di Asserzioni

Si può chiamare la funzione

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

per asserire (affermare) che i tipi `T1` e `T2` sono gli stessi. La funzione non fa nulla se lasserzione è soddisfatta. Se i tipi sono diversi, la chiamata alla funzione non verrà compilata, il messaggio di errore del compilatore dirà che `T1 e T2 non sono dello stesso tipo` e molto probabilmente (a seconda del compilatore) mostrerà i valori effettivi di `T1` e di `T2`. Ciò è utile principalmente allinterno del codice template.

Avvertenza: Se utilizzato allinterno di una funzione membro di una classe template o di una funzione template, `StaticAssertTypeEq<T1, T2>()` è efficace solo se viene istanziata la funzione. Ad esempio, dato:

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

il codice:

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

non genererà un errore del compilatore, poiché `Foo<bool>::Bar()` non viene mai effettivamente istanziato. Occorre, invece:

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

per causare un errore del compilatore.

8.2.9 Posizionamento delle Asserzioni

Le asserzioni si possono usare in qualsiasi funzione C++. In particolare, non deve essere un metodo della classe fixture del test. Lunico vincolo è che le asserzioni che generano un errore irreversibile (`FAIL*` e `ASSERT_*`) sono utilizzabili solo nelle funzioni che restituiscono void. Questa è una conseguenza del fatto che Google non utilizza le eccezioni. Inserendola in una funzione non void si otterrà un errore di compilazione confuso del tipo `"error: void value not ignored as it ought to be"` o `"cannot initialize return object of type 'bool' with an rvalue of type 'void'"` o `"error: no viable conversion from 'void' to 'string'"`.

Per usare asserzioni fatali in una funzione che restituisce non void, unopzione è fare in modo che la funzione restituisca il valore in un parametro di output. Per esempio, si può riscrivere `T2 Foo(T1 x)` in `void Foo(T1 x, T2* result)`. `*result` deve contenere un valore sensato anche quando la funzione ritorna prematuramente. Dato che la funzione ora restituisce `void`, si può utilizzare qualsiasi asserzione al suo interno.

Se non è possibile cambiare il tipo della funzione, si devono semplicemente usare asserzioni che generano errori non fatali, come `ADD_FAILURE*` e `EXPECT_*`.

{: .callout .note} NOTA: Costruttori e distruttori non sono considerati funzioni che restituiscono void, secondo le specifiche del linguaggio C++, e quindi non è possibile utilizzare asserzioni fatali al loro interno; provandoci si riceverà un errore di compilazione. Invece, o si chiama `abort` e si manda in crash tutto leseguibile del test, oppure si inserisce lasserzione fatale in una funzione `SetUp/TearDown`; vedere constructor/destructor vs. `SetUp/TearDown`

{: .callout .warning} ATTENZIONE: Unasserzione fatale in una funzione helper (metodo privato che restituisce void) chiamata da un costruttore o da un distruttore non termina il test corrente, come potrebbe suggerire lintuito: ritorna semplicemente dal costruttore o anticipatamente dal distruttore, forse lasciando loggetto in uno stato di parzialmente creato o di parzialmente distrutto! Quasi certamente si vuole, invece, `abort`ire o usare `SetUp/TearDown`.

8.3 Saltare lesecuzione del test

Per quanto riguarda le asserzioni `SUCCEED()` e `FAIL()`, si può evitare lesecuzione dei test successivi a runtime con la macro `GTEST_SKIP()`. Ciò è utile quando si devono verificare le precondizioni del sistema sotto test durante il runtime e saltare i test in modo significativo.

`GTEST_SKIP()` è utilizzabile nei casi di test individuali o nei metodi `SetUp()` di una classe derivata da `::testing::Environment` o da `::testing::Test`. Per esempio:

```
TEST(SkipTest, DoesSkip) {
  GTEST_SKIP() << "Skipping single test";
  FAIL();  // Won't fail; it won't be executed
}

class SkipFixture : public ::testing::Test {
 protected:
  void SetUp() override {
    GTEST_SKIP() << "Skipping all tests for this fixture";
  }
};

// Tests for SkipFixture won't be executed.
TEST_F(SkipFixture, SkipsOneTest) {
  FAIL();  // Won't fail; it won't be executed
}
```

Come con le macro delle asserzioni, si può accodare un messaggio personalizzato in `GTEST_SKIP()`.

8.4 Insegnare a GoogleTest Come Stampare i Propri Valori

Quando unasserzione di test come `EXPECT_EQ` fallisce, GoogleTest stampa i valori degli argomenti per eseguire il debug. Lo fa utilizzando una stampante con valori estensibile dallutente.

Questa stampante sa lavorare con i tipi nativi di C++, gli array, i contenitori STL e qualsiasi tipo che supporti loperatore «. Per gli altri tipi, stampa i byte grezzi nel valore e spera che lutente possa capirlo.

Come accennato in precedenza, la stampante è *extensible*. Ciò significa che le si può insegnare a stampare un proprio tipo particolare piuttosto che eseguirne il dump dei byte. Per farlo, si definisce un overload di `AbslStringify()` come una funzione template `friend` per il proprio tipo:

```
namespace foo {

class Point {  // We want GoogleTest to be able to print instances of this.
  ...
  // Provide a friend overload.
  template <typename Sink>
  friend void AbslStringify(Sink& sink, const Point& point) {
    absl::Format(&sink, "(%d, %d)", point.x, point.y);
  }

  int x;
  int y;
};

// If you can't declare the function in the class it's important that the
// AbslStringify overload is defined in the SAME namespace that defines Point.
// C++'s look-up rules rely on that.
enum class EnumWithStringify { kMany = 0, kChoices = 1 };
```

(continues on next page)

(continua dalla pagina precedente)

```
template <typename Sink>
void AbslStringify(Sink& sink, EnumWithStringify e) {
  absl::Format(&sink, "%s", e == EnumWithStringify::kMany ? "Many" : "Choices");
}

}  // namespace foo
```

{: .callout .note} Nota: `AbslStringify()` usa un generico buffer ńsinkż per costruire la sua stringa. For more information about supported operations on `AbslStringify()`s sink, see the `AbslStringify()` documentation.

`AbslStringify()` può anche usare lidentificatore di tipo ńtuttofareż `%v` di `absl::StrFormat`, allinterno delle proprie stringhe di formato per eseguire la deduzione del tipo. Il `Point` sopra potrebbe, per esempio, essere formattato come `"(%v, %v)"` e dedurre i valori `int` come `%d`.

A volte, `AbslStringify()` potrebbe non essere unopzione: il team potrebbe voler stampare tipi con informazioni di debug aggiuntive solo a scopo di test. Se è così, si può definire una funzione `PrintTo()` come questa:

```
#include <ostream>

namespace foo {

class Point {
  ...
  friend void PrintTo(const Point& point, std::ostream* os) {
    *os << "(" << point.x << "," << point.y << ")";
  }

  int x;
  int y;
};

// If you can't declare the function in the class it's important that PrintTo()
// is defined in the SAME namespace that defines Point.  C++'s look-up rules
// rely on that.
void PrintTo(const Point& point, std::ostream* os) {
    *os << "(" << point.x << "," << point.y << ")";
}

}  // namespace foo
```

Se è stato definito sia `AbslStringify()` che `PrintTo()`, questultimo verrà utilizzato da GoogleTest. Ciò consente di personalizzare il modo in cui il valore appare nelloutput di GoogleTest senza influenzare il codice che si basa sul comportamento di `AbslStringify()`.

Se si dispone di un operatore « esistente e si vuol definire un `AbslStringify()`, questultimo verrà utilizzato per la stampa di GoogleTest.

Per stampare in proprio un valore `x` utilizzando la stampante dei valori di GoogleTest, basta chiamare `::testing::PrintToString(x)`, che restituisce una `std::string`:

```
vector<pair<Point, int> > point_ints = GetPointIntVector();

EXPECT_TRUE(IsCorrectPointIntVector(point_ints))
    << "point_ints = " << testing::PrintToString(point_ints);
```

For more details regarding `AbslStringify()` and its integration with other libraries, see the documentation.

8.5 Sintassi delle Espressioni Regolari

Quando si builda con Bazel e utilizzando Abseil, GoogleTest usa la sintassi RE2. Otherwise, for POSIX systems (Linux, Cygwin, Mac), GoogleTest uses the POSIX extended regular expression syntax. Per conoscere la sintassi POSIX, si potrebbe leggere questa voce di Wikipedia.

Su Windows, GoogleTest utilizza la propria semplice implementazione di espressioni regolari. Mancano molte funzionalità. Ad esempio, non supportiamo lunione (`"x|y"`), il raggruppamento (`"(xy)"`), le parentesi quadre (`"[xy]"`) e il conteggio delle ripetizioni (`"x{5,7}"`), tra le altre cose. Di seguito è riportato ciò che supportiamo (`A` indica un carattere letterale, un punto (`.`), o una singola sequenza di escape `\\` ; `x` e `y` denotano espressioni regolari):

Espressione	Significato
`c`	corrisponde a qualsiasi carattere letterale `c`
`\\d`	corrisponde a qualsiasi cifra decimale
`\\D`	corrisponde a qualsiasi carattere che non sia una cifra decimale
`\\f`	corisponde a `\f`
`\\n`	corisponde a `\n`
`\\r`	corisponde a `\r`
`\\s`	corrisponde a qualsiasi spazio bianco ASCII, incluso `\n`
`\\S`	corrisponde a qualsiasi carattere che non sia uno spazio bianco
`\\t`	corisponde a `\t`
`\\v`	corisponde a `\v`
`\\w`	corrisponde a qualsiasi lettera, _, o cifra decimale
`\\W`	corrisponde a qualsiasi carattere che non corrisponde a `\\w`
`\\c`	corrisponde a qualsiasi carattere letterale `c`, che deve essere un segno di punteggiatura
`.`	corrisponde a qualsiasi carattere singolo tranne `\n`
`A?`	corrisponde a 0 o 1 occorrenze di `A`
`A*`	corrisponde a 0 o più occorrenze di `A`
`A+`	corrisponde a 0 o più occorrenze di `A`
`^`	corrisponde allinizio di una stringa (non a quello di ogni riga)
`$`	corrisponde alla fine di una stringa (non a quella di ogni riga)
`xy`	corrisponde a `x` seguito da `y`

Per aiutare a determinare quale funzionalità è disponibile sul sistema, GoogleTest definisce le macro per governare quale espressione regolare sta utilizzando. Le macro sono: `GTEST_USES_SIMPLE_RE=1` o `GTEST_USES_POSIX_RE=1`. Per far funzionare in ogni caso i death test, si può usare `#if` con queste macro o usare solo la sintassi più limitata.

8.6 I Death Test

In molte applicazioni sono presenti asserzioni che possono causare errori dellapplicazione se una condizione non viene soddisfatta. Questi controlli di coerenza, per verificare che il programma sia in uno stato sicuramente buono, sono destinati a fallire il prima possibile dopo che uno stato del programma viene corrotto. Se lasserzione verifica la condizione sbagliata, il programma potrebbe procedere in uno stato errato, che potrebbe portare al danneggiamento della memoria, a buchi di sicurezza o peggio. Pertanto è di vitale importanza verificare che tali affermazioni funzionino come previsto.

Since these precondition checks cause the processes to die, we call such tests *death tests*. Più in generale, qualsiasi test che controlli se un programma termini (tranne che per uneccezione) nel modo previsto è un ńdeath testż.

Si noti che se un pezzo di codice genera uneccezione, non la si considera ńmorteż ai fini dei ńdeath testż, poiché il chiamante del codice potrebbe rilevare leccezione ed evitare il crash. Per verificare le eccezioni generate dal codice, consultare *Exception Assertions*.

Per testare gli errori `EXPECT_*()/ASSERT_*()` nel codice, consultare *ńCatchingż degli Errori*.

8.6.1 Come Scrivere un Death Test

GoogleTest fornisce macro di asserzioni per supportare i death test. Vedere *Death Assertions* nei ńRiferimenti sulle Asserzioniż per i dettagli.

Per scrivere un death test, basta usare una delle macro nella funzione di test. Per esempio,

```
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, KillProcess) {
  EXPECT_EXIT(KillProcess(), testing::KilledBySignal(SIGKILL),
              "Sending myself unblockable signal");
}
```

verifica che:

- la chiamata a `Foo(5)` provoca la morte del processo con il messaggio di errore indicato,
- la chiamata a `NormalExit()` fa sì che il processo stampi `"Success"` su stderr e esca con il codice di uscita 0, e
- la chiamata a `KillProcess()` termina [kill] il processo con il segnale `SIGKILL`.

{: .callout .warning} Warning: If your death test contains mocks and is expecting a specific exit code, then you must allow the mock objects to be leaked via `Mock::AllowLeak`. This is because the mock leak detector will exit with its own error code if it detects a leak.

Se necessario, il corpo della funzione di test può contenere anche altre asserzioni e dichiarazioni.

Notare che un death test si preoccupa solo di tre cose:

1. con `listruzione` si abortisce o si esce dal processo?
2. (nel caso di `ASSERT_EXIT` e di `EXPECT_EXIT`) lo stato di uscita soddisfa il `predicato`? Oppure (nel caso di `ASSERT_DEATH` e di `EXPECT_DEATH`) lo stato di uscita è diverso da zero? E
3. loutput su stderr corrisponde al `matcher`?

In particolare, se `listruzione` genera un errore `ASSERT_*` o `EXPECT_*`, **non** farà fallire il death test, poiché le asserzioni di GoogleTest non interrompono il processo.

8.6.2 Nomenclatura del Death Test

{: .callout .important} IMPORTANTE: Consigliamo vivamente di seguire la convenzione di denominare la **test suite** (not il test) `*DeathTest` quando contiene un death test, come mostrato nellesempio precedente. La seguente sezione *I Death Test E i Thread* spiega il perché.

Se una classe fixture è condivisa tra test normali e death test, si può utilizzare `using` o `typedef` per introdurre un alias per la classe fixture ed evitare di duplicarne il codice:

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

using FooDeathTest = FooTest;
```

(continues on next page)

(continua dalla pagina precedente)

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

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

8.6.3 Come Funziona

Vedere *Death Assertions* nei ńRiferimenti sulle Asserzioniż.

8.6.4 I Death Test E i Thread

Il motivo dei due stili di death test ha a che fare con la ńthread safetyż. A causa dei noti problemi con il forking in presenza dei thread, i death test dovrebbero essere eseguiti in un contesto a thread singolo. A volte, però, non è possibile organizzare questo tipo di ambiente. Ad esempio, i moduli inizializzati staticamente possono avviare i thread prima che venga raggiunto il main. Una volta creati i thread, potrebbe essere difficile o impossibile ripulirli.

GoogleTest ha tre funzionalità destinate ad aumentare la consapevolezza sui problemi di threading.

1. Viene emesso un warning se sono in esecuzione più thread quando viene riscontrato un death test.
2. Le test suite con un nome che termina con ńDeathTestż vengono eseguite prima di tutti gli altri test.
3. Usa `clone()` invece di `fork()` per lo spawn del processo figlio su Linux (`clone()` non è disponibile su Cygwin né su Mac), in quanto `fork()` è più probabile che causi il blocco [hang] del figlio quando il processo genitore ha più thread.

È perfettamente corretto creare thread allinterno di una dichiarazione di death test; vengono eseguiti in un processo separato e non possono influenzare il genitore.

8.6.5 Stili dei Death Test

Lo stile ńthreadsafeż del death test è stato introdotto per mitigare i rischi dei test in un ambiente multithread. Scambia un aumento del tempo di esecuzione del test (potenzialmente in modo drammatico) con una migliore sicurezza del thread.

Il framework di test automatizzato non imposta il flag dello stile. Si può scegliere un particolare stile dei death test, impostando il flag a da programma:

```
GTEST_FLAG_SET(death_test_style, "threadsafe");
```

Lo si può fare nel `main()` per impostare lo stile per tutti i death test nel binario o nei singoli test. Ricordarsi che i flag vengono salvati prima di eseguire ciascun test e ripristinati alla fine, quindi non è necessario farlo esplicitamente. Per esempio:

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

TEST(MyDeathTest, TestOne) {
  GTEST_FLAG_SET(death_test_style, "threadsafe");
  // This test is run in the "threadsafe" style:
  ASSERT_DEATH(ThisShouldDie(), "");
}
```

(continues on next page)

(continua dalla pagina precedente)

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

8.6.6 Avvertenze

Largomento `statement` di `ASSERT_EXIT()` può essere qualsiasi istruzione C++ valida. Se lascia la funzione corrente tramite unistruzione `return` o per uneccezione, il death test viene considerato fallito. Alcune macro di GoogleTest potrebbero restituire dalla funzione corrente (ad esempio `ASSERT_TRUE()`), quindi sono da evitarle in `statement`.

Poiché `statement` viene eseguito nel processo figlio, qualsiasi effetto collaterale in memoria (ad esempio la modifica di una variabile, il rilascio di memoria, ecc.) da esso causato *non* sarà osservabile nel processo genitore. In particolare, se si rilascia memoria in un death test, il programma fallirà il controllo dellheap poiché il processo principale non vedrà mai la memoria recuperata. Per risolvere questo problema, si può

1. cercare di non liberare la memoria in un death test;
2. liberare nuovamente la memoria nel processo genitore; o
3. non utilizzare lheap checker nel programma.

A causa di un dettaglio implementativo, non è possibile inserire più asserzioni di death test sulla stessa riga; in caso contrario la compilazione fallirà con un messaggio di errore non evidente.

Nonostante la migliorata sicurezza dei thread offerta dallo stile del death test ńthreadsafeż, problemi dei thread come deadlock sono ancora possibili in presenza di gestori registrati con `pthread_atfork(3)`.

8.7 Uso delle Asserzioni nelle Sub-routine

{: .callout .note} Nota: Per inserire una serie di asserzioni di test in una subroutine per verificare una condizione complessa, considerare invece lutilizzo di un matcher GMock personalizzato. Ciò consente di fornire un messaggio di errore più leggibile in caso di errore ed evitare tutti i problemi descritti di seguito.

8.7.1 Aggiungere Trace alle Asserzioni

Se una sub-routine di test viene chiamata da più posti, quando unasserzione al suo interno fallisce, può essere difficile dire da quale invocazione della sub-routine provenga il fallimento. È possibile alleviare questo problema utilizzando log aggiuntivi o messaggi di errore personalizzati, ma ciò di solito è ingombrante per i test. Una soluzione migliore è quella di utilizzare la macro `SCOPED_TRACE` o lutility `ScopedTrace`:

```
SCOPED_TRACE(message);
```

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

dove `message` può essere qualsiasi cosa accodabile a `std::ostream`. La macro `SCOPED_TRACE` farà sì che il nome del file corrente, il numero di riga e il messaggio specificato vengano aggiunti in ogni messaggio di errore. `ScopedTrace` accetta nomi di file e numeri di riga espliciti negli argomenti, il che è utile per scrivere helper di test. Leffetto verrà annullato quando il controllo lascerà lo scope lessicale corrente.

Per esempio,

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

(continues on next page)

(continua dalla pagina precedente)

```
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: }
```

potrebbe generare messaggi come questi:

```
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
```

Senza il trace, sarebbe stato difficile sapere da quale invocazione di `Sub1()` provengono rispettivamente i due errori. (Si potrebbe aggiungere un messaggio extra a ciascuna asserzione in `Sub1()` per indicare il valore di `n`, ma è noioso).

Alcuni suggerimenti sullutilizzo di `SCOPED_TRACE`:

1. Con un messaggio adatto, spesso è sufficiente utilizzare `SCOPED_TRACE` allinizio di una subroutine, anziché in ciascuna chiamata.
2. Quando si chiamano subroutine allinterno di un ciclo, rendere literatore del ciclo parte del messaggio in `SCOPED_TRACE` in modo tale da poter sapere da quale iterazione proviene lerrore.
3. A volte il numero di riga del punto del trace è sufficiente per identificare la particolare invocazione di una subroutine. In questo caso, non è necessario scegliere un messaggio univoco per `SCOPED_TRACE`. Basta usare `""`.
4. Si può utilizzare `SCOPED_TRACE` in uno scope interno (annidato) quando ce nè uno nello scope esterno. In questo caso, tutti i trace point attivi verranno inclusi nei messaggi di errore, nellordine inverso in cui vengono incontrati.
5. Il trace dump è cliccabile in Emacs: premere `return` su un numero di riga e si passerà a quella riga nel file sorgente!

8.7.2 Propagazione degli Errori Fatali

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. Per esempio, il seguente test genererà un 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) {
```

(continues on next page)

(continua dalla pagina precedente)

```
  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!
}
```

Per alleviare questo problema, GoogleTest fornisce tre diverse soluzioni. Potresti utilizzare le eccezioni, le asserzioni `(ASSERT|EXPECT)_NO_FATAL_FAILURE` o la funzione `HasFatalFailure()`. Sono tutte descritte nelle due sottosezioni seguenti.

Asserzione sulle Subroutine con uneccezione

Il seguente codice può trasformare ASSERT-failure in uneccezione:

```
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();
}
```

Questo listener deve essere aggiunto dopo gli altri listener, se ce ne sono, altrimenti non vedranno `OnTestPartResult` fallire.

Asserzione sulle Subroutine

Come mostrato sopra, se il test chiama una subroutine che presenta un errore `ASSERT_*`, il test continuerà dopo il ritorno della subroutine. Questo potrebbe non essere quello che si vuole.

Spesso le persone vogliono che i fallimenti fatali si propaghino come le eccezioni. Per questo GoogleTest offre le seguenti macro:

Asserzione fatale	Asserzione non-fatale	Verifica
`ASSERT_NO_FATAL_FAILURE(s`	`EXPECT_NO_FATAL_FAILURE(s`	`statement` non genera nuovi errori fatali nel thread corrente.

Vengono controllati solo gli errori nel thread che esegue lasserzione per determinare il risultato di questo tipo di asserzioni. Se `statement` crea nuovi thread, gli errori in questi thread vengono ignorati.

Esempi:

```
ASSERT_NO_FATAL_FAILURE(Foo());

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

Le asserzioni provenienti da più thread non sono attualmente supportate in Windows.

Verifica degli Errori nel Test Corrente

`HasFatalFailure()` nella classe `::testing::Test` restituisce `true` se unasserzione nel test corrente ha subito un fallimento fatale. Ciò consente alle funzioni di rilevare errori fatali in una subroutine e di tornare anticipatamente.

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

Lutilizzo tipico, che sostanzialmente simula il comportamento della generazione di uneccezione, è:

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

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

Se `HasFatalFailure()` viene utilizzato al di fuori di `TEST()` , `TEST_F()` o di una test fixture, si deve aggiungere il prefisso `::testing::Test::`, come in:

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

Allo stesso modo, `HasNonfatalFailure()` restituisce `true` se il test corrente ha almeno un errore non fatale e `HasFailure()` restituisce `true` se il test corrente presenta almeno un errore di entrambi i tipi.

8.8 Log di Informazioni Aggiuntive

Nel codice del test, si può chiamare `RecordProperty("key", value)` per loggare informazioni aggiuntive, dove `value` può essere una stringa o un `int`. L*ultimo* valore registrato per una chiave verrà emesso nell*XML output* se se ne specifica uno. Per esempio, il test

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

restituirà XML in questo modo:

```
  ...
    <testcase name="MinAndMaxWidgets" file="test.cpp" line="1" status="run" time="0.
↪006" classname="WidgetUsageTest" MaximumWidgets="12" MinimumWidgets="9" />
  ...
```

{: .callout .note}

NOTA:

- `RecordProperty()` è un membro statico della classe `Test`. Pertanto deve avere il prefisso `::testing::Test::` se utilizzato al di fuori del corpo di `TEST` e della classe della fixture.
- *key* deve essere un nome di un attributo XML valido e non può entrare in conflitto con quelli già utilizzati da GoogleTest (`name`, `status`, `time`, `classname`, `type_param`, e `value_param`).
- È consentito chiamare `RecordProperty()` al di fuori della durata [lifespan] di un test. Se viene chiamato al di fuori di un test ma tra i metodi `SetUpTestSuite()` e `TearDownTestSuite()`

della test suite, verrà attribuito allelemento XML per la test suite. Se viene chiamato al di fuori di tutte le test suite (ad esempio in un ambiente di test), verrà attribuito allelemento XML di livello superiore.

8.9 Condivisione delle Risorse Tra Test nella Stessa Test Suite

GoogleTest crea un nuovo oggetto fixture per ogni test per rendere i test indipendenti e più facili da debuggare. Tuttavia, a volte i test utilizzano risorse costose da configurare, rendendo il modello ńone-copy-per-testż proibitivamente costoso.

Se i test non modificano la risorsa, non cè alcun danno nel condividere una singola copia della risorsa. Pertanto, oltre alla configurazione/dismissione per ogni test, GoogleTest supporta anche la configurazione/dismissione per ogni test suite. Per usarlo:

1. Nella classe della fixture (ad esempio `FooTest`), si dichiarano come `static` alcune variabili membro per contenere le risorse condivise.
2. Allesterno della classe fixture (in genere appena sotto di essa), si definiscono tali variabili membro, facoltativamente dando loro valori iniziali.
3. Nella stessa classe fixture, si definisce una funzione membro pubblica `static void SetUpTestSuite()` (ricordarsi di non scriverlo come **`SetupTestSuite`** con una u minuscola!) per impostare le risorse condivise e una funzione `static void TearDownTestSuite()` per eliminarle.

Questo è tutto! GoogleTest chiama automaticamente `SetUpTestSuite()` prima di eseguire il *primo test* nella test suite `FooTest` (ovvero prima di creare il primo oggetto `FooTest`) e chiama `TearDownTestSuite()` dopo aver eseguito l*ultimo test* al suo interno (ovvero dopo aver eliminato lultimo oggetto `FooTest`). Nel mezzo, i test possono utilizzare le risorse condivise.

Da ricordare che lordine dei test non è definito, quindi il codice non può dipendere da un test che ne precede o ne segue un altro. Inoltre, i test non devono modificare lo stato di alcuna risorsa condivisa oppure, se modificano lo stato, devono ripristinarlo al suo valore originale prima di passare il controllo al test successivo.

Notare che `SetUpTestSuite()` può essere chiamato più volte per una classe fixture che ha classi derivate, quindi non ci si deve aspettare che il codice nel corpo della funzione venga eseguito solo una volta. Inoltre, le classi derivate hanno ancora accesso alle risorse condivise definite come membri statici, quindi è necessaria unattenta considerazione quando si gestiscono le risorse condivise per evitare ńmemory leakż se le risorse condivise non vengono ripulite correttamente in `TearDownTestSuite()`.

Ecco un esempio di configurazione e dismissione ńper-test-suiteż:

```
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 ...;

    // If `shared_resource_` is **not deleted** in `TearDownTestSuite()`,
    // reallocation should be prevented because `SetUpTestSuite()` may be called
    // in subclasses of FooTest and lead to memory leak.
    //
    // if (shared_resource_ == nullptr) {
    //   shared_resource_ = new ...;
    // }
  }

  // Per-test-suite tear-down.
  // Called after the last test in this test suite.
  // Can be omitted if not needed.
```

(continues on next page)

(continua dalla pagina precedente)

```
  static void TearDownTestSuite() {
    delete shared_resource_;
    shared_resource_ = nullptr;
  }

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

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

  // 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 ...
}
```

{: .callout .note} NOTA: Sebbene il codice sopra dichiari `SetUpTestSuite()` protected, a volte potrebbe essere necessario dichiararlo pubblico, come quando lo si utilizza con `TEST_P`.

8.10 Set-Up e Tear-Down Globali

Proprio come è possibile eseguire limpostazione e la dismissione [tear-down] a livello di test e di test suite, è possibile farlo anche a livello di programma di test. Ecco come.

Per prima cosa, si crea una sottoclasse della classe `::testing::Environment` per definire un ambiente di test, che sappia come impostare e dismettere:

```
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 {}
};
```

Poi, si registra unistanza della classe di ambiente con GoogleTest chiamando la funzione `::testing::AddGlobalTestEnvironment()`:

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

Now, when `RUN_ALL_TESTS()` is invoked, it first calls the `SetUp()` method. The tests are then executed, provided that none of the environments have reported fatal failures and `GTEST_SKIP()` has not been invoked. Finally, `TearDown()` is called.

Note that `SetUp()` and `TearDown()` are only invoked if there is at least one test to be performed. Importantly, `TearDown()` is executed even if the test is not run due to a fatal failure or `GTEST_SKIP()`.

Calling `SetUp()` and `TearDown()` for each iteration depends on the flag `gtest_recreate_environments_when_repeating`. `SetUp()` and `TearDown()` are called for each environment object when the object is recreated for each iteration. However, if test environments are not recreated for each iteration, `SetUp()` is called only on the first iteration, and `TearDown()` is called only on the last iteration.

È consentito registrare più oggetti dellambiente. In questa suite, il loro `SetUp()` verrà chiamato nellordine in cui sono registrati, e il loro `TearDown()` verrà chiamato nellordine inverso.

Si noti che GoogleTest assume la proprietà degli oggetti dellambiente registrati. Pertanto **non vanno eliminati** manualmente.

Si deve chiamare `AddGlobalTestEnvironment()` prima di chiamare `RUN_ALL_TESTS()`, probabilmente in `main()`. Se si usa `gtest_main`, lo si deve chiamare prima dellavvio di `main()` affinché abbia effetto. Un modo per farlo è definire una variabile globale come questa:

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

Tuttavia, consigliamo vivamente di scrivere il `main()` e di chiamare `AddGlobalTestEnvironment()` lì, poiché fare affidamento sullinizializzazione delle variabili globali rende il codice più difficile da leggere e potrebbe causare problemi quando si registrano più ambienti da diverse unità di traduzione e gli ambienti hanno dipendenze tra loro (ricordare che il compilatore non garantisce lordine in cui vengono inizializzate le variabili globali di diverse unità di traduzione).

8.11 Test con Valori Parametrizzati

I *test con valori parametrizzati* consentono di testare il codice con parametri diversi senza scrivere più copie dello stesso test. Ciò è utile in diverse situazioni, ad esempio:

- Si ha una porzione di codice il cui comportamento è influenzato da uno o più flag della riga di comando. Si vuole essere sicuri che il codice funzioni correttamente per vari valori di questi flag.
- Si vogliono testare diverse implementazioni di uninterfaccia OO.
- Si vuol testare il codice su vari input (ovvero test basati sui dati [data-driven testing]). È facile abusare di questa funzione, quindi è meglio usare il buon senso!

8.11.1 Come Scrivere Test con Valori Parametrizzati

Per scrivere test con valori parametrizzati, è necessario innanzitutto definire una classe fixture. Deve essere derivata sia da `testing::Test` che da `testing::WithParamInterface<T>` (questultima è uninterfaccia pura), dove `T` è il tipo dei valori parametrizzati. Per comodità, si può semplicemente derivare la classe fixture da `testing::TestWithParam<T>`, che a sua volta è derivata sia da `testing::Test` che da `testing::WithParamInterface<T>`. `T` può essere qualsiasi tipo copiabile. Se si tratta di un puntatore semplice [raw], si è responsabili della gestione della durata dei valori puntati.

{: .callout .note} NOTA: Se la fixture definisce `SetUpTestSuite()` o `TearDownTestSuite()` devono essere dichiarati **public** anziché **protected** per poter utilizzare `TEST_P`.

```
class FooTest :
    public testing::TestWithParam<absl::string_view> {
  // 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,
```

(continues on next page)

(continua dalla pagina precedente)

```
                public testing::WithParamInterface<absl::string_view> {
  ...
};
```

Poi, si usa la macro `TEST_P` per definire tutti i pattern di test desiderati utilizzando questa fixture. Il suffisso `_P` sta per ńparametrizzatożo ńpatternż, a seconda di come si preferisce pensare.

```
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) {
  ...
}
```

Infine, si può utilizzare la macro `INSTANTIATE_TEST_SUITE_P` per istanziare la test suite con qualsiasi set di parametri desiderato. GoogleTest definisce una serie di funzioni per la generazione di parametri di test: vedere i dettagli in *`INSTANTIATE_TEST_SUITE_P`* nel ńRiferimento al Testingż.

Ad esempio, la seguente istruzione istanzia i test dalla test suite `FooTest` ciascuno con i valori dei parametrzzati `"meeny"`, `"miny"` e `"moe"` utilizzando il generatore di parametri *`Values`*:

```
INSTANTIATE_TEST_SUITE_P(MeenyMinyMoe,
                         FooTest,
                         testing::Values("meeny", "miny", "moe"));
```

{: .callout .note} NOTA: Il codice precedente deve essere inserito nello scope globale o del namespace, non nello scope della funzione.

Il primo argomento di `INSTANTIATE_TEST_SUITE_P` è un nome univoco per listanziazione della test suite. Largomento successivo è il nome del pattern di test e lultimo è il *generatore di parametri* [parameter generator].

Lespressione del generatore di parametri non viene valutata finché GoogleTest non viene inizializzato (tramite `InitGoogleTest()`). Qualsiasi inizializzazione precedente eseguita nella funzione `main` sarà accessibile dal generatore di parametri, ad esempio i risultati del parsing dei flag.

È possibile istanziare un pattern di test più di una volta, quindi per distinguere le diverse istanze del pattern, il nome di istanziazione viene aggiunto come prefisso a quello effettivo della test suite. Ricordarsi di scegliere prefissi univoci per le diverse istanze. I test dellistanziazione precedente avranno questi nomi:

- `MeenyMinyMoe/FooTest.DoesBlah/0` per `"meeny"`
- `MeenyMinyMoe/FooTest.DoesBlah/1` per `"miny"`
- `MeenyMinyMoe/FooTest.DoesBlah/2` per `"moe"`
- `MeenyMinyMoe/FooTest.HasBlahBlah/0` per `"meeny"`
- MeenyMinyMoe/FooTest.HasBlahBlah/1perżminyż`
- `MeenyMinyMoe/FooTest.HasBlahBlah/2` per `"moe"`

Si possono usare questi nomi in *`--gtest_filter`*.

La seguente istruzione istanzia nuovamente tutti i test da `FooTest`, ciascuno con i valori dei parametri `"cat"` e `"dog"` utilizzando il generatore di parametri *`ValuesIn`*:

```
constexpr absl::string_view kPets[] = {"cat", "dog"};
INSTANTIATE_TEST_SUITE_P(Pets, FooTest, testing::ValuesIn(kPets));
```

I test dellistanziazione precedente avranno questi nomi:

- `Pets/FooTest.DoesBlah/0` per `"cat"`
- `Pets/FooTest.DoesBlah/1` per `"dog"`
- `Pets/FooTest.HasBlahBlah/0` per `"cat"`
- `Pets/FooTest.HasBlahBlah/1` per `"dog"`

Si noti che `INSTANTIATE_TEST_SUITE_P` istanzia *tutti* i test nella test suite, indipendentemente dal fatto che le loro definizioni vengano prima o *dopo* la dichiarazione `INSTANTIATE_TEST_SUITE_P`.

Inoltre, per default, ogni `TEST_P` senza un corrispondente `INSTANTIATE_TEST_SUITE_P` provoca un test non riuscito nella test suite `GoogleTestVerification`. Se si dispone di una test suite in cui tale omissione non è un errore, ad esempio è in una libreria che potrebbe essere linkata per altri motivi o dove lelenco dei casi di test è dinamico e potrebbe essere vuoto, allora questo controllo può essere soppresso taggando la test suite:

```
GTEST_ALLOW_UNINSTANTIATED_PARAMETERIZED_TEST(FooTest);
```

Esaminare [sample7_unittest.cc] e [sample8_unittest.cc] per altri esempi.

8.11.2 Creazione di Test Astratti con Valori Parametrizzati

Abbiamo definito e istanziato `FooTest` nello *stesso* file sorgente. A volte si vorrebbero definire test con valori parametrizzati in una libreria e consentire ad altre persone di crearne unistanza successivamente. Questo pattern è detto *test astratti* [abstract tests]. Come esempio della sua applicazione, quando si progetta uninterfaccia è possibile scrivere una suite standard di test astratti (magari utilizzando una funzione factory come parametro di test) e ci si aspetta che tutte le implementazioni dellinterfaccia li superino. Quando qualcuno implementa linterfaccia, può creare unistanza della suite per ottenere gratuitamente tutti i test di conformità dellinterfaccia.

Per definire test astratti, si deve organizzare il codice in questo modo:

1. Si inserisce la definizione della classe della fixture parametrizzata (ad esempio `FooTest`) nel file header, ad esempio `foo_param_test.h`. Lo si consideri come una *dichiarazione* dei test astratti.
2. Si inseriscono le definizioni `TEST_P` in `foo_param_test.cc`, che include `foo_param_test.h`. Lo si consideri come *implementazione* dei test astratti.

Una volta definiti, se ne può creare unistanza includendo `foo_param_test.h`, richiamando `INSTANTIATE_TEST_SUITE_P()` e in base alla libreria target che contiene `foo_param_test.cc`. È possibile istanziare la stessa test suite astratta più volte, possibilmente in file sorgenti diversi.

8.11.3 Specificare i Nomi per i Parametri di test con Valori Parametrizzati

Lultimo argomento facoltativo di `INSTANTIATE_TEST_SUITE_P()` consente allutente di specificare una funzione o un funtore che genera suffissi personalizzati del nome del test in base ai parametri. La funzione deve accettare un argomento di tipo `testing::TestParamInfo<class ParamType>` e restituire `std::string`.

`testing::PrintToStringParamName` è un generatore di suffissi nativo che restituisce il valore di `testing::PrintToString(GetParam())`. Non funziona con `std::string` né con stringhe C.

{: .callout .note} NOTA: i nomi dei test devono essere non vuoti, univoci e possono contenere solo caratteri alfanumerici ASCII. In particolare, non devono contenere gli underscore

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

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

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

Fornire un funtore personalizzato consente un maggiore controllo sulla generazione dei nomi dei parametri dei test, in particolare per i tipi in cui la conversione automatica non genera nomi utili (ad esempio le stringhe come mostrato sopra). Lesempio seguente lo illustra per più parametri, un tipo di enumerazione e una stringa e mostra anche come combinare i generatori. Utilizza una lambda per concisione:

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

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

INSTANTIATE_TEST_SUITE_P(
    MyGroup, MyTestSuite,
    testing::Combine(
        testing::Values(MyType::MY_FOO, MyType::MY_BAR),
        testing::Values("A", "B")),
    [](const testing::TestParamInfo<MyTestSuite::ParamType>& info) {
      std::string name = absl::StrCat(
          std::get<0>(info.param) == MyType::MY_FOO ? "Foo" : "Bar",
          std::get<1>(info.param));
      absl::c_replace_if(name, [](char c) { return !std::isalnum(c); }, '_');
      return name;
    });
```

8.12 Test Tipizzati

Supponiamo di avere più implementazioni della stessa interfaccia e di voler assicurarci che tutte soddisfino alcuni requisiti comuni. Oppure si potrebbero aver definito diversi tipi che dovrebbero essere conformi allo stesso ńconceptż e si desidera verificarlo. In entrambi i casi, si desidera che la stessa logica di test venga ripetuta per tipi diversi.

Anche se si può scrivere un `TEST` o `TEST_F` per ogni tipo da testare (e si potrebbe anche fattorizzare la logica del test in una funzione template invocata da `TEST`), è noioso e non scalabile: se si vuole che `m` test su `n` tipi, si finirà per scrivere `m*n` `TEST`.

I *Test tipizzati* consentono di ripetere la stessa logica del test su un elenco di tipi. È necessario scrivere la logica del test solo una volta, anche se è necessario conoscere lelenco dei tipi quando si scrivono test tipizzati. Ecco come farlo:

Innanzitutto, si definisce una classe fixture template. Dovrebbe essere parametrizzata da un tipo. Ricordarsi di derivarlo da `::testing::Test`:

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

Poi, si associa un elenco di tipi alla test suite, che verrà ripetuto per ogni tipo nellelenco:

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

Lalias del tipo (`using` o `typedef`) è necessario affinché la macro `TYPED_TEST_SUITE` lo possa analizzare correttamente. Altrimenti il compilatore penserà che ogni virgola nellelenco dei tipi introduca un nuovo argomento della macro.

Poi, si usa `TYPED_TEST()` invece di `TEST_F()` per definire un test tipizzato per questa test suite. Lo si può ripetere più volte:

```
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) { ... }
```

Si può vedere [sample6_unittest.cc] per un esempio completo.

8.13 Test con i Tipi Parametrizzati

I *Type-parameterized tests* sono come quelli tipizzati, tranne per il fatto che non richiedono che si conosca in anticipo lelenco dei tipi. È invece possibile definire prima la logica del test e istanziarla successivamente con elenchi di tipi diversi. Si può anche crearne unistanza più di una volta nello stesso programma.

Se si sta progettando uninterfaccia o un concetto, si può definire una suite di test con i tipi parametrizzati per verificare le proprietà che dovrebbe avere qualsiasi implementazione valida dellinterfaccia/concetto. Quindi, lautore di ciascuna implementazione può semplicemente istanziare la test suite con il proprio tipo per verificare che sia conforme ai requisiti, senza dover scrivere ripetutamente test simili. Ecco un esempio:

Innanzitutto, definisce una classe fixture template, come abbiamo fatto con i test tipizzati:

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

Successivamente, si dichiara che si definirà una test suite con tipi parametrizzati:

```
TYPED_TEST_SUITE_P(FooTest);
```

Poi, si usa `TYPED_TEST_P()` per definire un test con un tipo parametrizzato. Lo si può ripetere più volte:

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

  // You will need to use `this` explicitly to refer to fixture members.
  this->DoSomethingInteresting()
  ...
}
```

(continues on next page)

(continua dalla pagina precedente)

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

Ora la parte difficile: si devono registrare tutti i pattern dei test utilizzando la macro `REGISTER_TYPED_TEST_SUITE_P` prima di poterli istanziare. Il primo argomento della macro è il nome della test suite; il resto sono i nomi dei test in questa test suite:

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

Infine, si può istanziare il pattern con i tipi desiderati. Se si inserisce il codice precedente in un file header, si può `#include`-rlo in più sorgenti C++e istanziarlo più volte.

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

Per distinguere le diverse istanze del pattern, il primo argomento della macro `INSTANTIATE_TYPED_TEST_SUITE_P` è un prefisso che verrà anteposto al nome effettivo della test suite. Ricordarsi di scegliere prefissi univoci per istanze diverse.

Nel caso speciale in cui lelenco dei tipi contiene solo un tipo, lo si può scrivere direttamente senza `::testing::Types<...>`, in questo modo:

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

Si può vedere [sample6_unittest.cc] per un esempio completo.

8.14 Test di Codice Privato

Se si modifica limplementazione interna del software, i test non dovrebbero smettere di funzionare fin quando la modifica non è osservabile dagli utenti. Pertanto, **secondo il principio del test della black-box, la maggior parte delle volte si deve testare il codice tramite le sue interfacce pubbliche**.

Se ci si ritrova ancora a dover testare il codice dellimplementazione interna, considerare se esiste un design migliore. Il desiderio di testare limplementazione interna è spesso un segno che la classe sta facendo troppo. Si prenda in considerazione lestrazione di una classe di implementazione e la si testi. Poi si usa quella classe di implementazione nella classe originale.

Se si deve assolutamente testare il codice dellinterfaccia non pubblica, lo si può fare. Ci sono due casi da considerare:

- Le funzioni statiche (*non* è lo stesso delle funzioni membro statiche!) o i namespace anonimi, e
- I membri di classe privati o protetti

Per testarli, utilizziamo le seguenti tecniche speciali:

- Sia le funzioni statiche che le definizioni/dichiarazioni in un namespace anonimo sono visibili solo allinterno della stessa unità di traduzione. To test them, 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. I file `.cc` di produzione e i test possono includere questo header interno, ma i non i clienti. In questo modo, si può testare completamente limplementazione interna senza divulgarla ai clienti.

{: .callout .note} NOTE: It is also technically *possible* to `#include` the entire `.cc` file being tested in your `*_test.cc` file to test static functions and definitions/declarations in an unnamed namespace. However, this technique is **not recommended** by this documentation and it is only presented here for the sake of completeness.

- I membri della classe privata sono accessibili solo dallinterno della classe o dai friend. Per accedere ai membri privati di una classe, si può dichiarare la fixture come friend della classe e definire come accedere nella fixture. I test che utilizzano la fixture possono poi accedere ai membri privati della classe di produzione

tramite gli accessori nella fixture. Notare che anche se la fixture è friend della classe di produzione, i test non lo sono automaticamente, poiché sono tecnicamente definiti in sottoclassi della fixture.

Un altro modo per testare i membri privati è quello di rifattorizzarli in una classe di implementazione, che viene poi dichiarata in un file `*-internal.h` file. I clienti non sono autorizzati a includere questo header, ma i test sì.

Oppure si può dichiarare un test individuale come friend della classe aggiungendo questa riga nel corpo della classe:

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

Per esempio,

```
// 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().
}
```

Prestare particolare attenzione quando la classe è definita in un namespace. Se si vuole che le fixture e i test siano friend della classe, allora devono essere definiti esattamente nello stesso namespace (nessun namespace anonimo o inline).

Ad esempio, se il codice da testare è simile a:

```
namespace my_namespace {

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

}  // namespace my_namespace
```

Il codice del test dovrebbe essere qualcosa del tipo:

```
namespace my_namespace {

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

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

}  // namespace my_namespace
```

8.15 ńCatchingż degli Errori

Se si crea unutilità di test al di sopra di GoogleTest, la si deve testare. Quale framework si userebbe per testarla? GoogleTest, ovviamente.

La sfida consiste nel verificare che lutilità di test riporti correttamente gli errori. Nei framework che riportano un errore lanciando uneccezione, si potrebbe catturare leccezione e metterci un assert. Ma GoogleTest non utilizza eccezioni, quindi come possiamo verificare che una parte di codice generi un errore previsto?

"gtest/gtest-spi.h" contiene alcuni costrutti per farlo. Dopo lo #include di questo header, si può usare

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

per asserire che `statement` genera un errore fatale (ad esempio `ASSERT_*`) nel thread corrente il cui messaggio contiene la `substring`, oppure si usa

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

se ci si aspetta un errore non fatale (per es. `EXPECT_*`).

Vengono controllati solo gli errori nel thread corrente per determinare il risultato di questo tipo di aspettative. Se `statement` crea nuovi thread, anche gli errori in questi thread vengono ignorati. Per rilevare errori anche in altri thread, si utilizza invece una delle seguenti macro:

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

{: .callout .note} NOTA: Le asserzioni provenienti da più thread non sono attualmente supportate su Windows.

Per ragioni tecniche, ci sono alcune avvertenze:

1. Non è possibile accodare a uno stream un messaggio di errore a nessuna delle macro.
2. `statement` in `EXPECT_FATAL_FAILURE{_ON_ALL_THREADS}()` non può fare riferimento a variabili locali non statiche o a membri non statici delloggetto `this`.
3. `statement` in `EXPECT_FATAL_FAILURE{_ON_ALL_THREADS}()` non può restituire un valore.

8.16 Registrazione dei test programmaticamente

Le macro `TEST` gestiscono la stragrande maggioranza di tutti i casi duso, ma ce ne sono alcuni in cui è richiesta la logica di registrazione a runtime. Per questi casi, il framework fornisce `::testing::RegisterTest` che consente ai chiamanti di registrare test arbitrari in modo dinamico.

Questa è unAPI avanzata da utilizzare solo quando le macro `TEST` sono insufficienti. Le macro dovrebbero essere preferite quando possibile, poiché evitano gran parte della complessità di chiamare questa funzione.

Fornisce la seguente firma [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);
```

Largomento `factory` è un oggetto richiamabile da factory (costruibile tramite spostamento) o un puntatore a funzione che crea una nuova istanza delloggetto Test. Gestisce la proprietà del chiamante. La firma del richiamabile è `Fixture*()`, dove `Fixture` è la classe fixture per il test. Tutti i test registrati con lo stesso `test_suite_name` devono restituire lo stesso tipo di fixture. Questo viene controllato in fase di esecuzione.

Il framework dedurrà la classe della fixture dalla factory e per questo chiamerà `SetUpTestSuite` e `TearDownTestSuite`.

Deve essere chiamato prima che venga invocato `RUN_ALL_TESTS()`, altrimenti il comportamento non è definito.

Esempio di caso duso:

```
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) {
  testing::InitGoogleTest(&argc, argv);
  std::vector<int> values_to_test = LoadValuesFromConfig();
  RegisterMyTests(values_to_test);
  ...
  return RUN_ALL_TESTS();
}
```

8.17 Recuperare il Nome del Test Corrente

A volte una funzione potrebbe aver bisogno di conoscere il nome del test attualmente in esecuzione. Ad esempio, si potrebbe utilizzare il metodo `SetUp()` della fixture per impostare il nome del ńgolden fileż (ndt: Output campione) in base al test in esecuzione. La classe *`TestInfo`* contiene queste informazioni.

Per ottenere un oggetto `TestInfo` per il test attualmente in esecuzione, si chiama `current_test_info()` sulloggetto singleton *`UnitTest`*:

```
  // 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()` restituisce un puntatore nullo se non è in esecuzione alcun test. In particolare, non è possibile trovare il nome della test suite in `SetUpTestSuite()`, `TearDownTestSuite()` (dove si conosce implicitamente il nome della test suite) né nelle funzioni chiamate da essi.

8.18 Estendere GoogleTest Gestendo gli Eventi dei Test

GoogleTest fornisce una **event listener API** per ricevere notifiche sullavanzamento di un programma di test e sugli errori dei test. Gli eventi che si possono rilevare [listen] includono, tra gli altri, linizio e la fine del programma di test, di una test suite o di un metodo di test. Questa API è utilizzabile per aumentare o sostituire loutput della console standard, sostituire loutput XML o fornire una forma di output completamente diversa, come una GUI o un database. È inoltre possibile utilizzare gli eventi dei test come checkpoint per implementare, ad esempio, un controllo delle perdite [leak] di risorse.

8.18.1 Definire i Listener degli Eventi

Per definire un event listener, si crea una sottoclasse di *`testing::TestEventListener`* o di *`testing::EmptyTestEventListener`*. La prima è uninterfaccia (astratta), in cui *ciascun metodo virtuale puro può essere sovrascritto [overridden] per gestire un evento di test* (Per esempio, quando inizia un test, verrà chiamato il metodo `OnTestStart()`). Lultimo fornisce unimplementazione vuota di tutti i metodi nellinterfaccia, in modo tale che una sottoclasse debba solo sovrascrivere i metodi che le interessano.

Quando viene generato un evento, il suo contesto viene passato alla funzione del gestore [handler] come argomento. Vengono utilizzati i seguenti tipi di argomento:

- UnitTest riflette lo stato dellintero programma di test,
- TestSuite contiene informazioni su una test suite, che può contenere uno o più test,
- TestInfo contiene lo stato di un test e
- TestPartResult rappresenta il risultato di unasserzione di un test.

Una funzione del gestore eventi può esaminare largomento che riceve per scoprire informazioni interessanti sullevento e sullo stato del programma di test.

Ecco un esempio:

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

  // Called after a failed assertion or a SUCCESS().
  void OnTestPartResult(const testing::TestPartResult& test_part_result) override {
    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.
  void OnTestEnd(const testing::TestInfo& test_info) override {
    printf("*** Test %s.%s ending.\n",
           test_info.test_suite_name(), test_info.name());
  }
};
```

8.18.2 Utilizzo dei Listener di Eventi

Per usare levent listener definito, se ne aggiunge unistanza alla lista degli event listener di GoogleTest (rappresentata dalla classe *TestEventListeners* - notare la ńsż alla fine del nome) nella funzione `main()`, prima di chiamare `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();
}
```

Cè solo un problema: la stampante di default dei risultati del test è ancora attiva, quindi il suo output si confonderà con quello della propria stampante minimalista. Per sopprimere la stampante di default , basta rilasciarla dallelenco del listener di eventi ed eseguirne il delete. Lo si può fare aggiungendo una riga:

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

Ora ci si può rilassare e godersi un risultato completamente diverso dai test. Per ulteriori dettagli, consultare [sample9_unittest.cc].

Si possono aggiungere più listener alla lista. Quando viene attivato un evento `On*Start()` o `OnTestPartResult()`, i listener lo riceveranno nellordine in cui appaiono nellelenco (poiché i nuovi listener vengono aggiunti alla finedelle lista, la stampante del testo e il generatore XML di default riceveranno per primi levento). Un evento `On*End()` verrà ricevuto dai listener nellordine *inverso*. Ciò consente alloutput dei listener aggiunti successivamente di essere inquadrati [framed] dalloutput dei listener aggiunti in precedenza.

8.18.3 Generare Errori nei Listener

Si possono utilizzare macro che generano errori (`EXPECT_*()`, `ASSERT_*()`, `FAIL()`, ecc.) durante lelaborazione di un evento. Ci sono alcune restrizioni:

1. Non è possibile generare alcun errore in `OnTestPartResult()` (altrimenti si chiamerà ricorsivamente `OnTestPartResult()`).
2. Un listener che gestisce `OnTestPartResult()` non può generare alcun errore.

Quando si aggiungono listener alla lista, si devono inserire i listener che gestiscono `OnTestPartResult()` *prima* di quelli che generano errori. Ciò garantisce che i fallimenti generati da questi ultimi vengano attribuiti al test corretto da parte dei primi.

Vedere [sample10_unittest.cc] per un esempio di un listener che genera errori.

8.19 Esecuzione dei Programmi di Test: Opzioni Avanzate

I programmi di test di GoogleTest sono normali eseguibili. Una volta creati, si possono eseguire direttamente e influenzarne il comportamento tramite le seguenti variabili di ambiente e/o flag della riga di comando. Affinché i flag funzionino, i programmi devono chiamare `::testing::InitGoogleTest()` prima di chiamare `RUN_ALL_TESTS()`.

Per visualizzare un elenco dei flag supportati e il loro utilizzo, eseguire il programma di test con il flag `--help`.

Se unopzione è specificata sia da una variabile dambiente che da un flag, questultimo ha la precedenza.

8.19.1 Selezione dei Test

Elenco dei Nomi dei Test

A volte è necessario elencare i test disponibili in un programma prima di eseguirli in modo da poter applicare un filtro, se necessario. Linclusione del flag `--gtest_list_tests` sovrascrive tutti gli altri flag ed elenca i test nel seguente formato:

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

Nessuno dei test elencati viene effettivamente eseguito se viene fornito il flag. Non esiste una variabile di ambiente corrispondente per questo flag.

Eseguire un Sottoinsieme dei Test

Per default, un programma GoogleTest esegue tutti i test definiti dallutente. A volte, se ne vogliono eseguire solo un sottoinsieme (ad esempio per eseguire il debug o verificare rapidamente una modifica). Se si imposta la variabile di ambiente `GTEST_FILTER` o il flag `--gtest_filter` su una stringa di filtro, GoogleTest eseguirà solo i test i cui nomi completi (nel formato `TestSuiteName.TestName`) corrisponderanno al filtro.

Il formato di un filtro è un elenco di pattern di carattery jolly, separati da `:` (detti *pattern positivi*) seguito facoltativamente da un `-` e un altro elenco di pattern separati da `:` (detti *pattern negativi*). Un test corrisponde al filtro se e solo se corrisponde a uno qualsiasi dei pattern positivi ma non corrisponde a nessuno dei pattern negativi.

Un pattern può contenere `'*'` (corrispondente a qualsiasi stringa) o `'?'` (corrispondente a qualsiasi carattere singolo). Per comodità, il filtro `'*-NegativePatterns'` può anche essere scritto come `'-NegativePatterns'`.

Per esempio:

- `./foo_test` Non ha flag e quindi esegue tutti i suoi test.
- `./foo_test --gtest_filter=*` Esegue tutto, a causa del singolo valore `*` corrispondente a tutto.
- `./foo_test --gtest_filter=FooTest.*` Esegue tutto nella test suite `FooTest` .
- `./foo_test --gtest_filter=*Null*:*Constructor*` Esegue qualsiasi test il cui nome completo contenga `"Null"` o `"Constructor"` .
- `./foo_test --gtest_filter=-*DeathTest.*` Esegue tutti i test non-death.
- `./foo_test --gtest_filter=FooTest.*-FooTest.Bar` Esegue tutto nella test suite `FooTest` tranne `FooTest.Bar`.
- `./foo_test --gtest_filter=FooTest.*:BarTest.*-FooTest.Bar:BarTest.Foo` Esegue tutto nella test suite `FooTest` tranne `FooTest.Bar` e tutto nella test suite `BarTest` tranne `BarTest.Foo`.

Interrompere lesecuzione del test al primo errore

Per default, un programma GoogleTest esegue tutti i test definiti dallutente. In alcuni casi (ad esempio sviluppo ed esecuzione iterativi di test) potrebbe essere auspicabile interrompere lesecuzione del test al primo fallimento (scambiando la maggiore latenza con la completezza). Se la variabile di ambiente `GTEST_FAIL_FAST` o il flag `--gtest_fail_fast` sono settati, lesecutore dei test si interromperà non appena viene rilevato il primo errore.

Disattivazione Temporanea dei Test

Se si ha un test non funzionante che non si può correggere subito, si può aggiungere il prefisso `DISABLED_` al nome. Ciò lo escluderà dallesecuzione. Questo è meglio che commentare il codice o usare `#if 0`, poiché i test disabilitati vengono comunque compilati (e quindi non ńmarcirannoż).

Per disabilitare tutti i test in una test suite, si può aggiungere `DISABLED_` allinizio del nome di ciascun test o in alternativa aggiungerlo allinizio del nome della test suite.

Ad esempio, i seguenti test non verranno eseguiti da GoogleTest, anche se verranno comunque compilati:

```
// 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) { ... }
```

{: .callout .note} NOTA: Questa funzione deve essere utilizzata solo come soluzione temporanea. Resta ancora correggere i test disabilitati in un secondo momento. Come promemoria, GoogleTest stamperà un banner che avvisa se un programma di test contiene test disabilitati.

{: .callout .tip} TIP: Si può facilmente contare il numero di test disabilitati con `grep`. Questo numero può essere utilizzato come parametro per migliorare la qualità del test.

Abilitazione Temporanea dei Test Disabilitati

Per includere test disabilitati nellesecuzione del test, è sufficiente richiamare il programma di test con il flag `--gtest_also_run_disabled_tests` o impostare la variabile dambiente `GTEST_ALSO_RUN_DISABLED_TESTS` su un valore diverso da `0`. Lo si può combinare col flag `--gtest_filter` per selezionare ulteriormente quali test disabilitati eseguire.

8.19.2 Enforcing Having At Least One Test Case

A not uncommon programmer mistake is to write a test program that has no test case linked in. This can happen, for example, when you put test case definitions in a library and the library is not marked as ńalways linkż.

To catch such mistakes, run the test program with the `--gtest_fail_if_no_test_linked` flag or set the `GTEST_FAIL_IF_NO_TEST_LINKED` environment variable to a value other than `0`. Now the program will fail if no test case is linked in.

Note that *any* test case linked in makes the program valid for the purpose of this check. In particular, even a disabled test case suffices.

8.19.3 Enforcing Running At Least One Test Case

In addition to enforcing that tests are defined in the binary with `--gtest_fail_if_no_test_linked`, it is also possible to enforce that a test case was actually executed to ensure that resources are not consumed by tests that do nothing.

To catch such optimization opportunities, run the test program with the `--gtest_fail_if_no_test_selected` flag or set the `GTEST_FAIL_IF_NO_TEST_SELECTED` environment variable to a value other than `0`.

A test is considered selected if it begins to run, even if it is later skipped via `GTEST_SKIP`. Thus, `DISABLED` tests do not count as selected and neither do tests that are not matched by `--gtest_filter`.

8.19.4 Ripetizione dei Test

Di tanto in tanto ci si imbatterà in un test il cui risultato è incostante. Forse fallirà solo 11% delle volte, rendendo piuttosto difficile riprodurre il bug in un debugger. Questa può essere uno dei principali motivi di frustrazione.

Il flag `--gtest_repeat` consente di ripetere più volte tutti i metodi di test (o quelli selezionati) in un programma. Si spera che un test instabile alla fine fallisca e dia la possibilità di eseguire il debug. Ecco come usarlo:

```
$ 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.
```

(continues on next page)

(continua dalla pagina precedente)

```
$ 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.
```

Se il programma di test contiene codice *set-up/tear-down globale*, verrà ripetuto anche in ogni iterazione, poiché potrebbe esserci qualche instabilità. Per evitare di ripetere il set-up/tear-down globale, si specifica `--gtest_recreate_environments_when_repeating=false`{.nowrap}.

Si può anche specificare il numero di ripetizioni impostando la variabile di ambiente `GTEST_REPEAT`.

8.19.5 Mischiare i Test

Si può specificare il flag `--gtest_shuffle` (o impostare la variabile di ambiente `GTEST_SHUFFLE` a `1`) per eseguire i test in un programma in ordine casuale. Questo aiuta a rivelare delle cattive dipendenze tra i test.

Per default, GoogleTest utilizza un seme casuale calcolato dallora corrente. Pertanto si riceverà ogni volta un ordine diverso. Loutput della console include il valore di inizializzazione casuale, in modo da poter riprodurre successivamente un errore relativo allordine. Per specificare esplicitamente il seed casuale, si usa il flag `--gtest_random_seed=SEED` (o si imposta la variabile di ambiente `GTEST_RANDOM_SEED`), dove `SEED` è un numero intero nellintervallo [0, 99999]. Il valore seed 0 è speciale: indica a GoogleTest di eseguire il comportamento di default di calcolare il seed dallora corrente.

Combinandolo con `--gtest_repeat=N`, GoogleTest selezionerà un seme casuale diverso e rimischierà i test in ogni iterazione.

8.19.6 Distribuire le Funzioni di Test su Più Macchine

Disponendo di più di macchine su cui eseguire un programma di test, si potrebbero eseguire le funzioni di test in parallelo e ottenere il risultato più velocemente. Chiamiamo questa tecnica *sharding*, dove ogni macchina è chiamata *shard*.

GoogleTest è compatibile con lo sharding dei test. Per sfruttare questa funzione, il test runner (non parte di GoogleTest) deve effettuare le seguenti operazioni:

1. Allocare un certo numero di macchine (shard) per eseguire i test.
2. Su ogni shard, si imposta la variabile di ambiente `GTEST_TOTAL_SHARDS` sul numero totale di shard. Deve essere lo stesso per tutti gli shard.
3. Su ogni shard, si imposta la variabile di ambiente `GTEST_SHARD_INDEX` sullindice dello shard. A shard diversi devono essere assegnati indici diversi, che devono essere compresi nellintervallo `[0, GTEST_TOTAL_SHARDS - 1]`.
4. Eseguire lo stesso programma di test su tutti gli shard. Quando GoogleTest vede le due variabili di ambiente precedenti, selezionerà un sottoinsieme delle funzioni di test da eseguire. In tutti gli shard, ciascuna funzione di test nel programma verrà eseguita esattamente una volta.
5. Attendi che tutti gli shard finiscano, quindi raccogliere e riportare i risultati.

Il progetto potrebbe contenere test scritti senza GoogleTest e quindi non capire questo protocollo. Affinché il test runner possa capire quale test supporta lo sharding, può impostare la variabile di ambiente `GTEST_SHARD_STATUS_FILE` su un path di un file inesistente. Se un programma di test supporta lo sharding, creerà questo file per riconoscere questo fatto; altrimenti non lo creerà. Il contenuto effettivo del file non è importante al momento, anche se potremmo inserirvi alcune informazioni utili in futuro.

Ecco un esempio per chiarire. Supponiamo di avere un programma di test `foo_test` che contiene le seguenti 5 funzioni di test:

```
TEST(A, V)
TEST(A, W)
TEST(B, X)
TEST(B, Y)
TEST(B, Z)
```

Supponiamo di avere 3 macchine a nostra disposizione. Per eseguire le funzioni di test in parallelo, si imposta GTEST_TOTAL_SHARDS a 3 su tutte le macchine, e si setta GTEST_SHARD_INDEX a 0, 1, e 2 sulle macchine rispettivamente. Poi si esegue lo stesso foo_test su ciascuna macchina.

GoogleTest si riserva il diritto di modificare la modalità di distribuzione del lavoro tra gli shard, ma ecco uno scenario possibile:

- La macchina #0 esegue A.V e B.X.
- La macchina #1 esegue A.W e B.Y.
- La macchina #2 esegue B.Z.

8.19.7 Controllo dellOutput del Test

Output Colorato del Terminale

GoogleTest può utilizzare i colori nelloutput del terminale per facilitare lindividuazione delle informazioni importanti:

Si può impostare la variabile di ambiente GTEST_COLOR o il flag della riga di comando --gtest_color a yes, no o auto (il default) per abilitare i colori, disabilitarli o lasciare che sia GoogleTest a decidere. Quando il valore è auto, GoogleTest utilizzerà i colori se e solo se loutput arriva a un terminale e (su piattaforme non Windows) la variabile di ambiente TERM è settata a xterm o a xterm-color.

Sopprimere i test superati

Per default, GoogleTest stampa 1 riga di output per ogni test, indicando se è stato superato o meno. Per mostrare solo i test falliti, eseguire il programma di test con --gtest_brief=1, o impostare la variabile di ambiente GTEST_BRIEF a 1.

Sopprimere il Tempo Trascorso

Per default, GoogleTest stampa il tempo necessario per eseguire ciascun test. Per disabilitarlo, si esegue il programma di test con il flag della riga di comando --gtest_print_time=0 o si setta la variabile di ambiente GTEST_PRINT_TIME a 0.

Sopprimere lOutput di Testo UTF-8

In caso di errori di asserzione, GoogleTest stampa i valori previsti ed effettivi di tipo string sia come stringhe con codifica esadecimale sia come testo UTF-8 leggibile se contengono caratteri UTF-8 non ASCII validi. Per sopprimere il testo UTF-8 perché, ad esempio, non si dispone di un supporto di output compatibile con UTF-8, si esegue il programma di test con --gtest_print_utf8=0 o si setta la variabile di ambiente GTEST_PRINT_UTF8 con 0.

Generare un Report XML

GoogleTest può emettere un report XML dettagliato in un file, oltre al normale output testuale. Il report contiene la durata di ciascun test e quindi può aiutare a identificare i test lenti.

Per generare il report XML, si imposta la variabile di ambiente GTEST_OUTPUT o il flag --gtest_output con la stringa "xml:path_to_output_file", che creerà il file nella posizione specificata. Si può anche utilizzare semplicemente la stringa "xml", nel qual caso loutput si trova nel file test_detail.xml nella directory corrente.

Specificando una directory (per esempio, "xml:output/directory/" su Linux o "xml:output\directory\" su Windows), GoogleTest creerà il file XML in tale directory, che prende il nome dalleseguibile del test (ad esempio foo_test.xml per il programma di test foo_test o foo_test.exe). Se il file esiste già (magari rimasto da

unesecuzione precedente), GoogleTest sceglierà un nome diverso (ad esempio `foo_test_1.xml`) per evitare di sovrascriverlo.

Il report si basa sul task Ant `junitreport`. Poiché tale formato era originariamente previsto per Java, è necessaria una piccola interpretazione per applicarlo ai test di GoogleTest, come mostrato qui:

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

- Lelemento root <testsuites> corrisponde allintero programma di test.
- Gli elementi <testsuite> corrispondono alle test suite di GoogleTest.
- Gli elementi <testcase> corrispondono alle funzioni di test di GoogleTest.

Ad esempio, il seguente programma

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

potrebbe generare questo report:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="3" failures="1" errors="0" time="0.035" timestamp="2011-10-
↪31T18:52:42" name="AllTests">
  <testsuite name="MathTest" tests="2" failures="1" errors="0" time="0.015">
    <testcase name="Addition" file="test.cpp" line="1" status="run" time="0.007"␣
↪classname="">
      <failure message="Value of: add(1, 1)&#x0A;  Actual: 3&#x0A;Expected: 2" type="
↪">...</failure>
      <failure message="Value of: add(1, -1)&#x0A;  Actual: 1&#x0A;Expected: 0" type="
↪">...</failure>
    </testcase>
    <testcase name="Subtraction" file="test.cpp" line="2" status="run" time="0.005"␣
↪classname="">
    </testcase>
  </testsuite>
  <testsuite name="LogicTest" tests="1" failures="0" errors="0" time="0.005">
    <testcase name="NonContradiction" file="test.cpp" line="3" status="run" time="0.
↪005" classname="">
    </testcase>
  </testsuite>
</testsuites>
```

Cose da notare:

- Lattributo `tests` di un elemento <testsuites> o <testsuite> indica quante funzioni di test contiene il programma GoogleTest o la test suite, mentre lattributo `failures` indica quanti di essi hanno fallito.
- Lattributo `time` esprime la durata del test, della test suite o dellintero programma di test in secondi.
- Lattributo `timestamp` registra la data e lora locale dellesecuzione del test.
- Gli attributi `file` e `line` registrano la posizione del file sorgente, dove è stato definito il test.

- Ogni elemento <failure> corrisponde a una singola asserzione GoogleTest non riuscita.

Generare un Report JSON

GoogleTest può anche emettere un report JSON come formato alternativo a XML. Per generare il report JSON, si imposta la variabile di ambiente `GTEST_OUTPUT` o il flag `--gtest_output` con la stringa `"json:path_to_output_file"`, che creerà il file nella posizione specificata. Si può anche utilizzare semplicemente la stringa `"json"`, nel qual caso loutput si trova nel file `test_detail.json` nella directory corrente.

Il formato del report è conforme al seguente schema JSON:

```
{
  "$schema": "https://json-schema.org/schema#",
  "type": "object",
  "definitions": {
    "TestCase": {
      "type": "object",
      "properties": {
        "name": { "type": "string" },
        "tests": { "type": "integer" },
        "failures": { "type": "integer" },
        "disabled": { "type": "integer" },
        "time": { "type": "string" },
        "testsuite": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/TestInfo"
          }
        }
      }
    },
    "TestInfo": {
      "type": "object",
      "properties": {
        "name": { "type": "string" },
        "file": { "type": "string" },
        "line": { "type": "integer" },
        "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" }
      }
```

(continues on next page)

(continua dalla pagina precedente)

```
    }
  },
  "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"
      }
    }
  }
}
```

Il report utilizza il formato conforme al seguente Proto3 utilizzando la codifica JSON:

```
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;
  string file = 6;
  int32 line = 7;
```

(continues on next page)

(continua dalla pagina precedente)

```
  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;
}
```

Ad esempio, il seguente programma

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

potrebbe generare questo 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",
          "file": "test.cpp",
          "line": 1,
          "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",
```

(continues on next page)

(continua dalla pagina precedente)

```
          "file": "test.cpp",
          "line": 2,
          "status": "RUN",
          "time": "0.005s",
          "classname": ""
        }
      ]
    },
    {
      "name": "LogicTest",
      "tests": 1,
      "failures": 0,
      "errors": 0,
      "time": "0.005s",
      "testsuite": [
        {
          "name": "NonContradiction",
          "file": "test.cpp",
          "line": 3,
          "status": "RUN",
          "time": "0.005s",
          "classname": ""
        }
      ]
    }
  ]
}
```

{: .callout .important} IMPORTANTE: Il formato esatto del documento JSON è soggetto a modifiche.

8.19.8 Controllare Come Vengono Riportati gli Errori

Rilevare le Uscite Premature dei Test

Google Test implements the *premature-exit-file* protocol for test runners to catch any kind of unexpected exits of test programs. Allavvio, Google Test crea il file che verrà automaticamente eliminato al termine di tutto il lavoro. Poi, il test runner può verificare se questo file esiste. Nel caso in cui il file rimanga non eliminato, il test ispezionato è terminato prematuramente.

Questa funzione è abilitata solo se è stata impostata la variabile di ambiente `TEST_PREMATURE_EXIT_FILE`.

Trasformare gli Errori delle Asserzioni in Break-Point

Quando si eseguono programmi di test in un debugger, è molto conveniente che il debugger possa rilevare un errore di asserzione e passare automaticamente alla modalità interattiva. La modalità *break-on-failure* di GoogleTest supporta questo comportamento.

Per abilitarlo, si imposta la variabile di ambiente `GTEST_BREAK_ON_FAILURE` su un valore diverso da `0`. In alternativa, si può utilizzare il flag della riga di comando `--gtest_break_on_failure`.

Disabilitare la Cattura di Eccezioni Test-Thrown

GoogleTest può essere utilizzato con o senza eccezioni abilitate. Se un test genera uneccezione C++ o (su Windows) uneccezione strutturata (SEH), per default GoogleTest la rileva, la segnala come un errore del test e continua con il metodo di test successivo. Ciò massimizza la copertura di unesecuzione di test. Inoltre, su Windows uneccezione non rilevata genererà una finestra popup, quindi catturare le eccezioni consente di eseguire i test automaticamente.

Durante il debug degli errori di test, tuttavia, si potrebbe invece volere che le eccezioni vengano gestite dal debugger, in modo da poter esaminare lo stack delle chiamate quando viene generata uneccezione. Per rag-

giungere questo obiettivo, si imposta la variabile di ambiente `GTEST_CATCH_EXCEPTIONS` su `0` o si usa il flag `--gtest_catch_exceptions=0` durante lesecuzione dei test.

8.19.9 Integrazione del Sanitizer

Undefined Behavior Sanitizer, Address Sanitizer e Thread Sanitizer forniscono tutti funzioni deboli che si possono sovrascrivere [override] per sollevare errori espliciti quando rilevano errori di sanitizer, come la creazione di un riferimento da `nullptr`. Per sovrascrivere queste funzioni, si inseriscono le relative definizioni in un file sorgente che si compila come parte del binario principale:

```
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"
```

Dopo aver compilato il progetto con uno dei sanitizer abilitati, se un particolare test attiva un errore del sanitizer, GoogleTest segnalerà che non è riuscito.

CAPITOLO 9

gMock per Principianti

9.1 Cosè gMock?

Quando si scrive un prototipo o un test, spesso non è fattibile o saggio affidarsi interamente a oggetti reali. Un **oggetto mock** implementa la stessa interfaccia di un oggetto reale (quindi può essere utilizzato in sua vece), ma consente di specificare in fase di esecuzione come verrà utilizzato e cosa dovrebbe fare (quali metodi saranno chiamati? In quale ordine? Quante volte? Con quali argomenti? Cosa restituiranno? ecc.).

È facile confondere il termine *oggetti fake* (falsi) con oggetto mock (simulati). I fake e i mock in realtà significano cose molto diverse nella comunità del Test-Driven Development (TDD):

- Gli oggetti **fake** hanno implementazioni funzionanti, ma solitamente prendono qualche scorciatoia (magari per rendere le operazioni meno costose), che li rende non adatti alla produzione. Un file system in memoria è un esempio di fake.
- I **mocks** sono oggetti preprogrammati con delle *expectation*, che formano una specifica delle chiamate che si aspettano di ricevere.

Se tutto questo appare troppo astratto, non cè da preoccuparsi: la cosa più importante da ricordare è che un mock consente di verificare l*interazione* tra se stesso e il codice che lo utilizza. La differenza tra i fake e i mock diventerà molto più chiara una volta che si inizia ad usare utilizzare i mock.

gMock è una libreria (a volte la chiamiamo anche ńframeworkż per farlo sembrare interessante) per creare classi mock e utilizzarle. Fa a C++ quello che jMock/EasyMock fa a Java (beh, più o meno).

Quando si usa gMock,

1. per prima cosa si usano alcune semplici macro per descrivere linterfaccia che si vuol simulare [mock] e queste si espanderanno nellimplementazione della classe mock;
2. successivamente, si creano alcuni oggetti mock e se ne specificano le aspettative e il comportamento utilizzando una sintassi intuitiva;
3. quindi si usa il codice che utilizza gli oggetti mock. gMock rileverà qualsiasi violazione delle aspettative non appena si presentano.

9.2 Perché gMock?

Sebbene gli oggetti mock aiutino a rimuovere le dipendenze non necessarie nei test e a renderli veloci e affidabili, usare i mock manualmente in C++ è *difficile*:

- Qualcuno deve implementare i mock. Il lavoro è solitamente noioso e soggetto a errori. Non cè da stupirsi che si facciano lunghi giri per evitarlo.
- La qualità di questi mock scritti manualmente è un po, ehm, imprevedibile. Se ne potrebbero vedere alcuni davvero raffinati, ma ce ne sono anche alcuni che sono stati fatti in fretta e hanno tutti i tipi di restrizioni ad hoc.
- La conoscenza acquisita utilizzando un mock non viene trasferita a quello successivo.

Al contrario, i programmatori Java e Python dispongono di alcuni ottimi framework di simulazione (jMock, EasyMock, ecc.), che automatizzano la creazione di mock. Di conseguenza, il mocking è una tecnica comprovata ed efficace e una pratica ampiamente adottata in quelle comunità. Avere lo strumento giusto fa assolutamente la differenza.

gMock è stato creato per aiutare i programmatori C++. È stato ispirato da jMock e EasyMock, ma progettato pensando alle specifiche del C++. Risulta amichevole se uno qualsiasi dei seguenti problemi è un disturbo:

- Si è bloccati con un progetto non ottimale e si vorrebbe aver fatto più prototipi prima che fosse troppo tardi, ma la prototipazione in C++ non è affatto ńrapidaż.
- I test sono lenti poiché dipendono da troppe librerie o utilizzano risorse costose (ad esempio un database).
- I test sono fragili poiché alcune risorse che utilizzano sono inaffidabili (ad esempio la rete).
- Si vuol testare il modo in cui il codice gestisce un errore (ad esempio un errore di checksum del file), ma non è facile causarne uno.
- Si vuol essere certi che un modulo interagisca con gli altri moduli nel modo giusto, ma è difficile osservare linterazione; quindi si ricorre allosservazione degli effetti collaterali alla fine dellazione, ma nella migliore delle ipotesi è imbarazzante.
- Si vogliono ńsimulareż [mock] le dipendenze, tranne per il fatto che non ci sono ancora implementazioni mock; e, francamente, non si è entusiasti di alcune di quelle scritte a mano.

Invitiamo a utilizzare gMock come

- uno strumento di *design*, poiché consente di sperimentare subito e spesso il design dellinterfaccia. Più iterazioni portano a progetti migliori!
- uno strumento di *test* per ridurre le dipendenze dagli output dei test e sondare linterazione tra il modulo e i suoi collaboratori.

9.3 Iniziamo

gMock è allegato a googletest.

9.4 Un Caso per Tartarughe Mock

Diamo unocchiata a un esempio. Supponiamo che si stia sviluppando un programma di grafica che si basa su unAPI simile a LOGO per il disegno. Come provare che faccia la cosa giusta? Bene, lo si può eseguire e confrontare lo schermo con una schermata campione, ma ammettiamolo: test come questo sono costosi da eseguire e fragili (e se si passasse a una nuova brillante scheda grafica con un anti-aliasing migliore? Allimprovviso si devono aggiornare tutte le immagini campione). Sarebbe troppo laborioso se tutti i test fossero così. Fortunatamente, è nota la Dependency Injection e si conosce la cosa giusta da fare: anziché far dialogare lapplicazione direttamente con lAPI di sistema, si avvolgono le API in uninterfaccia (ad esempio, `Turtle`) e il codice di tale interfaccia:

```
class Turtle {
  ...
  virtual ~Turtle() {}
  virtual void PenUp() = 0;
  virtual void PenDown() = 0;
  virtual void Forward(int distance) = 0;
  virtual void Turn(int degrees) = 0;
```

(continues on next page)

(continua dalla pagina precedente)

```
  virtual void GoTo(int x, int y) = 0;
  virtual int GetX() const = 0;
  virtual int GetY() const = 0;
};
```

(Notare che il distruttore di `Turtle` **deve** essere virtual, come nel caso di **tutte** le classi da cui si intende ereditare - altrimenti il distruttore della classe derivata non verrà chiamata quando si elimina un oggetto tramite un puntatore base e si otterranno stati del programma danneggiati come i memory leak).

Si può controllare se il movimento della tartaruga lascerà una traccia usando `PenUp()` e `PenDown()` e controllarne il movimento con `Forward()`, `Turn()` e `GoTo()`. Infine, `GetX()` e `GetY()` dicono la posizione attuale della tartaruga.

Il programma normalmente utilizzerà unimplementazione reale di questa interfaccia. Nei test è invece possibile utilizzare unimplementazione mock. Ciò consente di controllare facilmente quali primitive di disegno sta chiamando il programma, con quali argomenti e in quale ordine. I test scritti in questo modo sono molto più robusti (non si romperanno perché la nuova macchina esegue lanti-aliasing in modo diverso), sono più facili da leggere e mantenere (lintento di un test è espresso nel codice, non in alcune immagini binarie) e si gira *molto, molto più velocemente*.

9.5 Scrittura della Classe Mock

Se si è fortunati, i mock da utilizzare sono già stati implementati da alcune persone simpatiche. Se, tuttavia, ci si trova nella posizione di scrivere una classe mock, si può stare tranquilli: gMock trasforma questo compito in un gioco divertente! (Be quasi).

9.5.1 Come la si Definisce

Utilizzando linterfaccia di `Turtle` come esempio, ecco i semplici passaggi da seguire:

- Derivare una classe `MockTurtle` da `Turtle`.
- Prendere una funzione *virtual* di `Turtle` (sebbene sia possibile ńmock-areż metodi non-virtuali utilizzando i template, è molto più complicato).
- Nella sezione `public:` della classe figlia, si scrive `MOCK_METHOD();`
- Ora arriva la parte divertente: si prende la firma [signature] della funzione, la si taglia e la si incolla nella macro e si aggiungono due virgole: una tra il tipo restituito e il nome, unaltra tra il nome e lelenco degli argomenti.
- Per mock-are un metodo const, si aggiunge un 4ř parametro contenente `(const)` (le parentesi sono obbligatorie).
- Poiché si sta sovrascrivendo [overriding] un metodo virtuale, suggeriamo di aggiungere la parola chiave `override`. Per i metodi const il 4ř parametro diventa `(const, override)`, per i metodi non-const basta `(override)`. Questo non è obbligatorio.
- Si ripete per tutte le funzioni virtuali da simulare (mock-are). (Inutile dire che *tutti* i metodi virtuali puri nella classe astratta devono essere mock-ati o sovrascritti).

Alla fine del processo, si dovrebbe avere qualcosa del genere:

```
#include <gmock/gmock.h>  // Brings in gMock.

class MockTurtle : public Turtle {
 public:
  ...
  MOCK_METHOD(void, PenUp, (), (override));
  MOCK_METHOD(void, PenDown, (), (override));
  MOCK_METHOD(void, Forward, (int distance), (override));
```

(continues on next page)

(continua dalla pagina precedente)

```
  MOCK_METHOD(void, Turn, (int degrees), (override));
  MOCK_METHOD(void, GoTo, (int x, int y), (override));
  MOCK_METHOD(int, GetX, (), (const, override));
  MOCK_METHOD(int, GetY, (), (const, override));
};
```

Non è necessario definire questi metodi mock da qualche altra parte: la macro `MOCK_METHOD` genererà le definizioni automaticamente. È così semplice!

9.5.2 Dove Metterlo

Quando si definisce una classe mock, si deve decidere dove inserire la sua definizione. Qualcuno le mette in un `_test.cc`. Questo va bene quando linterfaccia mock-ata (ad esempio, `Foo`) è di proprietà della stessa persona o team. Altrimenti, quando il proprietario di `Foo` la modifica, il test potrebbe non funzionare. (Non ci si può certo aspettare che il manutentore di `Foo` corregga ogni test che usa `Foo`, vero?)

In generale, non si dovrebbero mock-are le classi altrui. Per mock-are una classe simile di altri, si definisce la classe mock nel pacchetto Bazel di `Foo` (di solito la stessa directory o una sottodirectory di `testing`) e la si inserisce in un `.h` e un `cc_library` con `testonly=True`. Dopodiché tutti possono farvi riferimento dai loro test. Se `Foo` cambia, cè solo una copia di `MockFoo` da modificare e solo i test che dipendono dai metodi modificati devono essere corretti.

Cè un altro modo per farlo: si può introdurre un sottile layer `FooAdaptor` al di sopra di `Foo` e inserire il codice per questa nuova interfaccia. Poiché si è proprietari di `FooAdaptor`, si possono assorbire più facilmente le modifiche in `Foo`. Anche se inizialmente questo richiede più lavoro, scegliere attentamente linterfaccia delladattatore [adaptor] può rendere il codice più facile da scrivere e più leggibile (un vantaggio netto a lungo termine), poiché si può scegliere `FooAdaptor` per adattarlo maggiormente al dominio specifico meglio di `Foo`.

9.6 Uso dei Mock nei Test

Una volta che si ha una classe mock, usarla è facile. Il flusso di lavoro tipico è:

1. Si importano i nomi gMock dal namespace `testing` in modo da poterli utilizzare senza qualificarli (lo si deve fare solo una volta per file). Ricordarsi che i namespace sono una buona idea.
2. Si creano degli oggetti mock.
3. Si specificano le [expectation] aspettative su di essi (quante volte verrà chiamato un metodo? Con quali argomenti? Cosa dovrebbe fare? ecc.).
4. Si esegue il codice che utilizza i mock; facoltativamente, si controlla il risultato utilizzando le asserzioni di googletest. Se un metodo mock viene chiamato più volte del previsto o con argomenti sbagliati, si riceverà immediatamente un errore.
5. Quando un mock viene distrutto, gMock controllerà automaticamente se tutte le expectation su di esso sono state soddisfatte.

Ecco un esempio:

```
#include "path/to/mock-turtle.h"
#include <gmock/gmock.h>
#include <gtest/gtest.h>

using ::testing::AtLeast;                         // #1

TEST(PainterTest, CanDrawSomething) {
  MockTurtle turtle;                              // #2
  EXPECT_CALL(turtle, PenDown())                  // #3
      .Times(AtLeast(1));
```

(continues on next page)

(continua dalla pagina precedente)

```
  Painter painter(&turtle);                     // #4

  EXPECT_TRUE(painter.DrawCircle(0, 0, 10));    // #5
}
```

Come intuito, questo test verifica che `PenDown()` venga chiamato almeno una volta. Se loggetto `painter` non ha chiamato questo metodo, il test fallirà con un messaggio come questo:

```
path/to/my_test.cc:119: Failure
Actual function call count doesn't match this expectation:
Actually: never called;
Expected: called at least once.
Stack trace:
...
```

Tip 1: Se si esegue il test da un buffer Emacs, si può premere <Invio> sul numero di riga per passare direttamente alla expectation non riuscita.

Tip 2: Se gli oggetti mock non vengono mai eliminati, la verifica finale non avrà luogo. Pertanto è una buona idea attivare il controllo dellheap nei test quando si allocano i mock sullheap. Lo si ottiene automaticamente se si usa già la libreria `gtest_main`.

9.6.1 Expectation Ordering

Nota importante: gMock richiede che le expectation siano impostate **prima** di chiamare le funzioni mock, altrimenti il comportamento è **indefinito**. Non alternare le chiamate a `EXPECT_CALL()` e le chiamate alle funzioni mock e non impostare alcuna expectation su un mock dopo averlo passato a unAPI.

Ciò significa che `EXPECT_CALL()` dovrebbe essere letto come se si aspettasse che una chiamata avverrà *in futuro*, non che una chiamata sia avvenuta. Perché gMock funziona così? Ebbene, specificare in anticipo la expectation consente a gMock di riportare una violazione non appena si verifica, quando il contesto (stack trace, ecc.) è ancora disponibile. Ciò semplifica molto il debug.

Certo, questo test è artificioso e non fa molto. Si può facilmente ottenere lo stesso effetto senza utilizzare gMock. Tuttavia, come vedremo presto, gMock permette di fare *molto di più* con i mock.

9.7 Impostare le Expectation

La chiave per utilizzare con successo un oggetto mock è quella di impostarvi le *expectation giuste*. Con delle expectation troppo rigide, il test fallirà per modifiche non correlate. Se sono troppo blande, qualche bug potrebbe sfuggire. Lo si vuole fare nel modo giusto in modo che il test possa rilevare esattamente il tipo di bug che si intende rilevare. gMock fornisce i mezzi necessari per farlo ńnel modo giustoż.

9.7.1 Sintassi Generale

In gMock usiamo la macro `EXPECT_CALL()` per impostare una expectation su un metodo mock. La sintassi generale è:

```
EXPECT_CALL(mock_object, method(matchers))
    .Times(cardinality)
    .WillOnce(action)
    .WillRepeatedly(action);
```

La macro ha due argomenti: prima loggetto mock, poi il metodo e i suoi argomenti. Si noti che i due sono separati da una virgola (`,`), non da un punto (`.`). (Perché si usa una virgola? La risposta è che era necessario per motivi tecnici). Se il metodo non è overloaded, la macro può essere richiamata anche senza i matcher:

```
EXPECT_CALL(mock_object, non-overloaded-method)
    .Times(cardinality)
    .WillOnce(action)
    .WillRepeatedly(action);
```

Questa sintassi consente a chi scrive il test di specificare che è ńchiamato con qualsiasi argomentoż senza specificare esplicitamente il numero o il tipo di argomenti. Per evitare ambiguità indesiderate, questa sintassi può essere utilizzata solo per metodi che non sono sovraccaricati [overloaded].

Entrambe le forme della macro possono essere seguite da alcune *clausole* [clause] facoltative che forniscono ulteriori informazioni sulla expectation. Illustreremo come funziona ciascuna clausola nelle prossime sezioni.

Questa sintassi è progettata per far sì che una expectation venga letta come linglese. Ad esempio, probabilmente si intuirà che

```
using ::testing::Return;
...
EXPECT_CALL(turtle, GetX())
    .Times(5)
    .WillOnce(Return(100))
    .WillOnce(Return(150))
    .WillRepeatedly(Return(200));
```

dice che il metodo `turtle` delloggetto `GetX()` verrà chiamato cinque volte, restituirà 100 la prima volta, 150 la seconda volta e poi 200 ogni volta. A qualcuno piace chiamare questo stile di sintassi un Domain-Specific Language (DSL).

{: .callout .note} **Nota:** Perché utilizziamo una macro per fare questo? Beh, ha due scopi: in primo luogo rende le expectation facilmente identificabili (sia da `grep` che da un lettore umano), e in secondo luogo consente a gMock di includere la posizione del file sorgente di una expectation non riuscita nei messaggi, facilitando il debugging.

9.7.2 I Matcher: Quali Argomenti Ci Aspettiamo?

Quando una funzione mock accetta argomenti, possiamo specificare quali ci aspettiamo, ad esempio:

```
// Expects the turtle to move forward by 100 units.
EXPECT_CALL(turtle, Forward(100));
```

Spesso non si vuol essere troppo specifici. Ricordate quando si parlava di test troppo rigidi? Una specifica eccessiva porta a test fragili e oscura lintento dei test. Pertanto invitiamo a specificare solo ciò che è necessario, né più né meno. Se non interessa il valore di un argomento, si scrive `_` come argomento, che significa ńva bene qualsiasi cosaż:

```
using ::testing::_;
...
// Expects that the turtle jumps to somewhere on the x=50 line.
EXPECT_CALL(turtle, GoTo(50, _));
```

`_` è unistanza di quelle che chiamiamo **matcher**. Un matcher è come un predicato e può verificare se un argomento è quello che ci aspetteremmo. Si può utilizzare un matcher allinterno di `EXPECT_CALL()` ovunque sia previsto un argomento di funzione. `_` è un modo conveniente per dire ńqualsiasi valoreż.

Negli esempi precedenti, anche `100` e `50` sono matcher; implicitamente, sono lo stesso di `Eq(100)` e `Eq(50)`, che specificano che largomento deve essere uguale (usando `operator==`) allargomento matcher. Esistono molti *matcher nativi [built-in]* per i tipi comuni (così come matcher personalizzati); per esempio:

```
using ::testing::Ge;
...
// Expects the turtle moves forward by at least 100.
EXPECT_CALL(turtle, Forward(Ge(100)));
```

Se non interessa *qualsiasi* argomento, anziché specificare _ per ciascuno di essi si può omettere lelenco dei parametri:

```
// Expects the turtle to move forward.
EXPECT_CALL(turtle, Forward);
// Expects the turtle to jump somewhere.
EXPECT_CALL(turtle, GoTo);
```

Funziona con tutti i metodi non-overloaded; se un metodo lo è, è necessario aiutare gMock a risolvere quale overload ci si aspetta specificando il numero di argomenti ed eventualmente anche i tipi degli argomenti.

9.7.3 Le Cardinalità: Quante Volte Verrà Chiamata?

La prima clausola che possiamo specificare dopo una `EXPECT_CALL()` è `Times()`. Chiamiamo il suo argomento una **cardinalità** per dire *quante volte* dovrebbe verificarsi la chiamata. Consente di ripetere una expectation molte volte senza in realtà scriverla tante volte. Ancora più importante, una cardinalità può essere ńfuzzyż (vago), proprio come può esserlo un matcher. Ciò consente allutente di esprimere esattamente lintento di un test.

Un caso speciale interessante è quando diciamo `Times(0)`. Come si intuisce - significa che la funzione non dovrebbe essere chiamata con gli argomenti forniti e gMock segnalerà un errore di googletest ogni volta che la funzione viene chiamata (erroneamente).

Abbiamo visto `AtLeast(n)` come esempio di cardinalità fuzzy in precedenza. Per lelenco delle cardinalità native utilizzabili, vedere qui.

La clausola `Times()` può essere omessa. **Se si omette `Times()`, gMock dedurrà autonomamente la cardinalità**. Le regole sono facili da ricordare:

- Se non cè **né** `WillOnce()` **né** `WillRepeatedly()` nella `EXPECT_CALL()`, la cardinalità dedotta è `Times(1)`.
- Se ci sono *n* `WillOnce()` ma **nessun** `WillRepeatedly()`, dove *n* >= 1, la cardinalità è `Times(n)`.
- Se ci sono *n* `WillOnce()` e **un** `WillRepeatedly()`, dove *n* >= 0, la cardinalità è `Times(AtLeast(n))`.

Quiz veloce: cosa accadrà se si prevede che una funzione venga chiamata due volte ma in realtà viene chiamata quattro volte?

9.7.4 Azioni: Cosa Dovrebbe Fare?

Ci si ricorda che un oggetto mock non ha in realtà unimplementazione funzionante? Noi come utenti dobbiamo dirgli cosa fare quando viene invocato un metodo. Questo è facile in gMock.

Innanzitutto, se il tipo restituito di una funzione mock è un tipo nativo o un puntatore, la funzione ha un**azione di default** (una funzione `void` tornerà e basta, una funzione `bool` restituirà `false` e le altre funzioni restituiranno 0). Inoltre, in C++ 11 e nelle versioni successive, una funzione mock il cui tipo restituito è default-constructible (ovvero ha un costruttore di default) ha unazione di default che restituisce un valore costruito dal default. Se non si dice nulla, verrà utilizzato questo comportamento.

In secondo luogo, se una funzione mock non ha unazione di default, o questa non è adatta alle proprie esigenze, si può specificare lazione da intraprendere ogni volta che la expectation corrisponde utilizzando una serie di clausole `WillOnce()` seguite da un facoltativo `WillRepeatedly()`. Per esempio,

```
using ::testing::Return;
...
EXPECT_CALL(turtle, GetX())
    .WillOnce(Return(100))
    .WillOnce(Return(200))
    .WillOnce(Return(300));
```

dice che `turtle.GetX()` verrà chiamata *esattamente tre volte* (gMock lo ha dedotto da quante clausole `WillOnce()` abbiamo scritto, poiché non abbiamo scritto esplicitamente `Times()`), e restituirà rispettivamente 100, 200 e 300.

```
using ::testing::Return;
...
EXPECT_CALL(turtle, GetY())
     .WillOnce(Return(100))
     .WillOnce(Return(200))
     .WillRepeatedly(Return(300));
```

dice che `turtle.GetY()` sarà chiamata *almeno due volte* (gMock lo sa perché abbiamo scritto due clausole `WillOnce()` e una `WillRepeatedly()` pur non avendo esplicitato `Times()`), restituirà 100 e 200 rispettivamente le prime due volte e 300 dalla terza volta in poi.

Naturalmente, se si scrive esplicitamente un `Times()`, gMock non tenterà di dedurre la cardinalità. Cosa succede se il numero specificato è maggiore delle clausole `WillOnce()`? Bene, dopo che tutti i `WillOnce()` sono esauriti, gMock eseguirà ogni volta lazione di *default* per la funzione (a meno che, ovviamente, non si abbia un `WillRepeatedly()`).

Cosa possiamo fare allinterno di `WillOnce()` oltre a `Return()`? Si può restituire un riferimento utilizzando `ReturnRef(variable)`, o richiamare una funzione pre-definita, tra [others] (altre).

Nota importante Listruzione `EXPECT_CALL()` valuta la clausola dellazione una sola volta, anche se lazione può essere eseguita più volte. Pertanto è necessario fare attenzione agli effetti collaterali. Quanto segue potrebbe non fare quello che si desidera:

```
using ::testing::Return;
...
int n = 100;
EXPECT_CALL(turtle, GetX())
    .Times(4)
    .WillRepeatedly(Return(n++));
```

Invece di restituire 100, 101, 102, , consecutivamente, questa funzione mock restituirà sempre 100 in quanto `n++` viene valutato solo una volta. Allo stesso modo, `Return(new Foo)` creerà un nuovo oggetto `Foo` quando viene eseguito `EXPECT_CALL()` e restituirà ogni volta lo stesso puntatore. Se si vuole che leffetto collaterale si verifichi ogni volta, si deve definire unazione personalizzata, che illustreremo nel cook book.

È ora di un altro quiz! Cosa significa quanto segue?

```
using ::testing::Return;
...
EXPECT_CALL(turtle, GetY())
    .Times(4)
    .WillOnce(Return(100));
```

Ovviamente `turtle.GetY()` ci si aspetta che venga chiamato quattro volte. Ma se si crede che restituirà 100 ogni volta, è meglio pensarci due volte! Ricordarsi che una clausola `WillOnce()` verrà utilizzata ogni volta che la funzione viene invocata e successivamente verrà eseguita lazione di default. Quindi la risposta giusta è che `turtle.GetY()` restituirà 100 la prima volta, ma **restituirà 0 dalla seconda volta in poi**, poiché restituire 0 è lazione di default per le funzioni `int`.

9.7.5 Uso di Expectation Multiple {#MultiExpectations}

Finora abbiamo mostrato solo esempi con una sola expectation. Più realisticamente, si specificheranno le expectation su più metodi mock che potrebbero provenire da più oggetti mock.

Per default, quando viene richiamato un metodo mock, gMock cercherà le expectation nell**ordine inverso** in cui sono definite e si fermerà quando viene trovata una expectation attiva che corrisponde agli argomenti (si possono considerare come ńle regole più recenti scavalcano quelle più vecchież). Se la corrispondente expectation non può accettare più chiamate, si verificherà un errore con violazione del limite superiore [upper-bound-violated]. Ecco un esempio:

```
using ::testing::_;
...
EXPECT_CALL(turtle, Forward(_));  // #1
EXPECT_CALL(turtle, Forward(10))  // #2
    .Times(2);
```

Se `Forward(10)` viene chiamato tre volte di seguito, la terza volta si verificherà un errore, poiché lultima expectation corrispondente (#2) è stata saturata. Se, tuttavia, la terza chiamata `Forward(10)` viene sostituita da `Forward(20)`, allora andrebbe bene, poiché ora #1 sarà la expectation corrispondente.

{: .callout .note} **Nota:** Perché gMock cerca una corrispondenza nellordine *inverso* delle expectation? Il motivo è che ciò consente allutente di impostare le expectation di default nel costruttore di un oggetto mock o nella fase di impostazione del [test fixture] e quindi personalizzare il mock expectation più specifiche nel corpo del test. Quindi, se si hanno due expectation sullo stesso metodo, si vuol mettere quella con matcher più specifici **dopo** laltra, altrimenti la regola più specifica verrebbe oscurata da quella più generale che viene dopo.

{: .callout .tip} **Tip:** È molto comune iniziare con una expectation generale per un metodo e con `Times(AnyNumber())` (omettendo gli argomenti, o con _ per tutti gli argomenti, se [overloaded] sovraccaricati). Ciò rende expected tutte le chiamate al metodo. Ciò non è necessario per i metodi non menzionati (questi sono ńpoco interessanti [uninteresting]ż), ma è utile per i metodi che hanno alcune expectation, ma per i quali vanno bene altre chiamate. Consultare Le Chiamate Poco Interessanti e Quelle Unexpected.

9.7.6 Chiamate ordinate e Non {#OrderedCalls}

Per default, una expectation può corrispondere a una chiamata anche se una expectation precedente non è stata soddisfatta. In altre parole, le chiamate non devono avvenire nellordine in cui sono specificate le expectation.

A volte, si vuole che tutte le chiamate previste avvengano in un ordine rigoroso. Dirlo in gMock è semplice:

```
using ::testing::InSequence;
...
TEST(FooTest, DrawsLineSegment) {
  ...
  {
    InSequence seq;

    EXPECT_CALL(turtle, PenDown());
    EXPECT_CALL(turtle, Forward(100));
    EXPECT_CALL(turtle, PenUp());
  }
  Foo();
}
```

Creando un oggetto di tipo `InSequence`, tutte le expectation nel suo scope vengono inserite in una *sequenza* e devono verificarsi *sequenzialmente*. Dato che facciamo affidamento solo sul costruttore e sul distruttore di questo oggetto per svolgere il lavoro vero e proprio, il suo nome è davvero irrilevante.

In questo esempio, testiamo che `Foo()` chiami le tre funzioni expected nellordine in cui sono scritte. Se una chiamata viene effettuata fuori ordine, sarà un errore.

(E se interessa lordine relativo di alcune chiamate, ma non di tutte? È possibile specificare un arbitrario ordine parziale? La risposta è sì! I dettagli si trovano qui.)

9.7.7 Tutte le expectation sono fisse (a meno che non venga detto diversamente) {#StickyExpectations}

Ora facciamo un breve quiz per vedere quanto bene si possano già usare queste cose di mock. Come verificare che alla tartaruga venga chiesto di andare allorigine *esattamente due volte* (ignorare qualsiasi altra istruzione ricevuta)?

Dopo aver trovato la risposta, date unocchiata alla nostra e confrontate le note (da risolvere in autonomia - non imbrogliare!):

```
using ::testing::_;
using ::testing::AnyNumber;
...
EXPECT_CALL(turtle, GoTo(_, _))  // #1
     .Times(AnyNumber());
EXPECT_CALL(turtle, GoTo(0, 0))  // #2
     .Times(2);
```

Supponiamo che `turtle.GoTo(0, 0)` sia chiamata tre volte. Nella terza volta, gMock vedrà che gli argomenti corrispondono alla expectation #2 (ricordate che scegliamo sempre lultima expectation corrispondente). Ora, poiché abbiamo detto che dovrebbero esserci solo due chiamate di questo tipo, gMock segnalerà immediatamente un errore. Questo è fondamentalmente ciò che abbiamo detto nella sezione *Uso di Expectation Multiple* sopra.

Questo esempio mostra che **le expectation in gMock sono per default ńappiccicoseż**, nel senso che rimangono attive anche dopo aver raggiunto i limiti superiori della loro invocazione. Questa è una regola importante da ricordare, poiché influenza il significato delle specifiche ed è **diversa** da come viene eseguita in molti altri framework di simulazione (Perché dovremmo farlo? Perché pensiamo che la nostra regola renda i casi comuni più facili da esprimere e comprendere).

Semplice? Vediamo se è stato ben compreso: cosa dice il seguente codice?

```
using ::testing::Return;
...
for (int i = n; i > 0; i--) {
  EXPECT_CALL(turtle, GetX())
      .WillOnce(Return(10*i));
}
```

Se si pensa che dica che `turtle.GetX()` verrà chiamato `n` volte e restituirà 10, 20, 30, , consecutivamente, ripensateci! Il problema è che, come abbiamo detto, le expectation sono appiccicose. Pertanto, la seconda volta che viene chiamato `turtle.GetX()`, lultima (più recente) istruzione `EXPECT_CALL()` corrisponderà e porterà immediatamente a un errore di ńupper bound violatedż: questo pezzo di codice non è molto utile!

Un modo corretto per dire che `turtle.GetX()` restituirà 10, 20, 30, , è dire esplicitamente che le expectation *non* sono appiccicose. In altre parole, dovrebbero *ritirarsi* appena saturate:

```
using ::testing::Return;
...
for (int i = n; i > 0; i--) {
  EXPECT_CALL(turtle, GetX())
      .WillOnce(Return(10*i))
      .RetiresOnSaturation();
}
```

E cè un modo migliore per farlo: in questo caso, ci aspettiamo che le chiamate avvengano in un ordine specifico e allineiamo le azioni in modo che corrispondano allordine. Poiché in questo caso lordine è importante, dovremmo renderlo esplicito utilizzando una sequenza:

```
using ::testing::InSequence;
using ::testing::Return;
...
{
  InSequence s;

  for (int i = 1; i <= n; i++) {
    EXPECT_CALL(turtle, GetX())
        .WillOnce(Return(10*i))
        .RetiresOnSaturation();
```

(continues on next page)

(continua dalla pagina precedente)

```
  }
}
```

A proposito, laltra situazione in cui una expectation può *non* essere appiccicosa è quando è in una sequenza: non appena viene utilizzata unaltra expectation che viene dopo di essa nella sequenza, viene automaticamente ritirata (e non verrà mai più usata per abbinare alcuna chiamata).

9.7.8 Chiamate Non Interessanti

Un oggetto mock può avere molti metodi e non tutti sono così interessanti. Ad esempio, in alcuni test potremmo non interessarci a quante volte `GetX()` e `GetY()` vengono chiamate.

In gMock, se non si è interessati a un metodo, non si dice nulla a riguardo. Se cè una chiamata a questo metodo, verrà visualizzato un warning nelloutput del test, ma non si tratterà di un errore. Questo comportamento è detto ńnaggyż (fastidioso); per modificarlo, vedere Nice, Strict e Naggy.

CAPITOLO 10

Ricettario di gMock

Qui si trovano delle ricette per utilizzare gMock. Se non è stato ancora fatto, leggere prima la guida gMock per Principianti per capire le basi.

{: .callout .note} **Nota:** gMock risiede nel namespace `testing`. Per maggiore leggibilità, si consiglia di scrivere `using ::testing::Foo;` una volta nel file prima di utilizzare il nome `Foo` definito da gMock. In questa sezione, per brevità, omettiamo le istruzioni `using`, ma nel codice lo si dovrebbe fare.

10.1 Creare Classi Mock

Le classi mock vengono definite come le normali classi, utilizzando la macro `MOCK_METHOD` per generare metodi mock-ati. La macro prende 3 o 4 parametri:

```
class MyMock {
 public:
  MOCK_METHOD(ReturnType, MethodName, (Args...));
  MOCK_METHOD(ReturnType, MethodName, (Args...), (Specs...));
};
```

I primi 3 parametri sono semplicemente la dichiarazione del metodo, divisa in 3 parti. Il 4ř parametro accetta un elenco chiuso di qualificatori, che riguardano il metodo generato:

- **`const`** - Rende il metodo mock-ato un metodo `const`. Obbligatorio se si ńoverrideż un metodo `const`.
- **`override`** - Contrassegna il metodo con `override`. Consigliato se si sovrascrive un metodo `virtual`.
- **`noexcept`** - Contrassegna il metodo con `noexcept`. Obbligatorio se si sovrascrive un metodo `noexcept`.
- **`Calltype(...)`** - Imposta il tipo di chiamata per il metodo (ad esempio su `STDMETHODCALLTYPE`), utile in Windows.
- **`ref(...)`** - Contrassegna il metodo con la qualifica del riferimento specificata. Obbligatorio se si esegue loverride di un metodo che dispone di qualifiche dei riferimenti. Ad esempio `ref(&)` o `ref(&&)`.

10.1.1 Trattare le virgole non protette

Le virgole non-protette, ovvero le virgole non racchiuse tra parentesi, impediscono a `MOCK_METHOD` di analizzare correttamente i suoi argomenti:

{: .bad}

```
class MockFoo {
 public:
  MOCK_METHOD(std::pair<bool, int>, GetPair, ());  // Won't compile!
  MOCK_METHOD(bool, CheckMap, (std::map<int, double>, bool));  // Won't compile!
};
```

Soluzione 1 - racchiudere tra le parentesi:

{: .good}

```
class MockFoo {
 public:
  MOCK_METHOD((std::pair<bool, int>), GetPair, ());
  MOCK_METHOD(bool, CheckMap, ((std::map<int, double>), bool));
};
```

Si noti che racchiudere un tipo di ritorno o un tipo di un argomento tra le parentesi non è, in generale, sbagliato in C++. `MOCK_METHOD` rimuove le parentesi.

Soluzione 2 - definire un alias:

{: .good}

```
class MockFoo {
 public:
  using BoolAndInt = std::pair<bool, int>;
  MOCK_METHOD(BoolAndInt, GetPair, ());
  using MapIntDouble = std::map<int, double>;
  MOCK_METHOD(bool, CheckMap, (MapIntDouble, bool));
};
```

10.1.2 Il Mock di Metodi Privati or Protetti

Si deve sempre inserire una definizione del metodo mock (`MOCK_METHOD`) in una sezione `public:` della classe mock, indipendentemente dal fatto che il metodo mocked sia `public`, `protected`, o `private` nella classe base. Ciò consente a `ON_CALL` e a `EXPECT_CALL` di fare riferimento alla funzione mock dallesterno della classe mock. (Sì, il C++ consente a una sottoclasse di modificare il livello di accesso di una funzione virtuale nella classe base). Esempio:

```
class Foo {
 public:
  ...
  virtual bool Transform(Gadget* g) = 0;

 protected:
  virtual void Resume();

 private:
  virtual int GetTimeOut();
};

class MockFoo : public Foo {
 public:
  ...
  MOCK_METHOD(bool, Transform, (Gadget* g), (override));

  // The following must be in the public section, even though the
  // methods are protected or private in the base class.
  MOCK_METHOD(void, Resume, (), (override));
```

(continues on next page)

(continua dalla pagina precedente)

```
  MOCK_METHOD(int, GetTimeOut, (), (override));
};
```

10.1.3 Mock di Metodi Overloaded

Si possono avere funzioni overloaded mock-ate come al solito. Non è richiesta alcuna attenzione particolare:

```
class Foo {
  ...

  // Must be virtual as we'll inherit from Foo.
  virtual ~Foo();

  // Overloaded on the types and/or numbers of arguments.
  virtual int Add(Element x);
  virtual int Add(int times, Element x);

  // Overloaded on the const-ness of this object.
  virtual Bar& GetBar();
  virtual const Bar& GetBar() const;
};

class MockFoo : public Foo {
  ...
  MOCK_METHOD(int, Add, (Element x), (override));
  MOCK_METHOD(int, Add, (int times, Element x), (override));

  MOCK_METHOD(Bar&, GetBar, (), (override));
  MOCK_METHOD(const Bar&, GetBar, (), (const, override));
};
```

{: .callout .note} **Nota:** se non si mock-ano tutte le versioni del metodo sovraccaricato, il compilatore darà un warning sul fatto che che alcuni metodi nella classe base sono nascosti. Per risolvere questo problema, usare `using` per inserirli nello scope:

```
class MockFoo : public Foo {
  ...
  using Foo::Add;
  MOCK_METHOD(int, Add, (Element x), (override));
  // We don't want to mock int Add(int times, Element x);
  ...
};
```

10.1.4 Mock di Classi Template

Si può avere la versione mock di classi template come con qualsiasi classe.

```
template <typename Elem>
class StackInterface {
  ...
  // Must be virtual as we'll inherit from StackInterface.
  virtual ~StackInterface();

  virtual int GetSize() const = 0;
  virtual void Push(const Elem& x) = 0;
};
```

(continues on next page)

(continua dalla pagina precedente)

```
template <typename Elem>
class MockStack : public StackInterface<Elem> {
  ...
  MOCK_METHOD(int, GetSize, (), (const, override));
  MOCK_METHOD(void, Push, (const Elem& x), (override));
};
```

10.1.5 Mock di Metodi Non-virtual {#MockingNonVirtualMethods}

gMock può produrre versioni mock di funzioni non-virtual da utilizzare nellinserimento [injection] delle dipendenze Hi-perf.

In questo caso, invece di condividere una classe base comune con la classe reale, la classe mock sarà *non correlata* alla classe reale, ma conterrà metodi con le stesse firme. La sintassi per il mock dei metodi non-virtual è la *stessa* dei mock dei metodi virtual (basta non aggiungere `override`):

```
// A simple packet stream class.  None of its members is virtual.
class ConcretePacketStream {
 public:
  void AppendPacket(Packet* new_packet);
  const Packet* GetPacket(size_t packet_number) const;
  size_t NumberOfPackets() const;
  ...
};

// A mock packet stream class.  It inherits from no other, but defines
// GetPacket() and NumberOfPackets().
class MockPacketStream {
 public:
  MOCK_METHOD(const Packet*, GetPacket, (size_t packet_number), (const));
  MOCK_METHOD(size_t, NumberOfPackets, (), (const));
  ...
};
```

Notare che la classe mock non definisce `AppendPacket()`, come la classe reale. Va bene fin quando non è necessario che il test lo chiami.

Successivamente, cè bisogno di un modo per dire che si vuole utilizzare `ConcretePacketStream` nel codice di produzione mentre `MockPacketStream` nei test. Dato che le funzioni non sono virtuali e le due classi non sono correlate, si deve specificare la scelta in *fase di compilazione* (invece che in fase di esecuzione [run time]).

Un modo per farlo è creare un template del codice che deve utilizzare lo stream di packet. Più specificamente, si fornirà al codice un argomento di tipo template per il tipo di stream di packet. In produzione, si creerà unistanza del template con `ConcretePacketStream` come argomento del tipo. Nei test, si istanzierà lo stesso template con `MockPacketStream`. Per esempio, si può scrivere:

```
template <class PacketStream>
void CreateConnection(PacketStream* stream) { ... }

template <class PacketStream>
class PacketReader {
 public:
  void ReadPackets(PacketStream* stream, size_t packet_num);
};
```

Poi si possono usare `CreateConnection<ConcretePacketStream>()` e `PacketReader<ConcretePacketStream>` nel codice di produzione, e nei test `CreateConnection<MockPacketStream>()` e `PacketReader<MockPacketStream>`.

```
  MockPacketStream mock_stream;
  EXPECT_CALL(mock_stream, ...)...;
  .. set more expectations on mock_stream ...
  PacketReader<MockPacketStream> reader(&mock_stream);
  ... exercise reader ...
```

10.1.6 Mock di Funzioni Libere [Free]

Non è possibile avere direttamente mock di una funzione libera (ad esempio una funzione in stile C o un metodo statico). Se necessario, si può riscrivere il codice per utilizzare uninterfaccia (classe astratta).

Invece di chiamare direttamente una funzione libera (ad esempio, `OpenFile`), se ne introduce uninterfaccia e si dispone una sottoclasse concreta che chiama la funzione libera:

```
class FileInterface {
 public:
  ...
  virtual bool Open(const char* path, const char* mode) = 0;
};

class File : public FileInterface {
 public:
  ...
  bool Open(const char* path, const char* mode) override {
     return OpenFile(path, mode);
  }
};
```

Il codice dovrebbe comunicare con `FileInterface` per aprire un file. Ora è facile mock-are la funzione.

Potrebbe sembrare una seccatura, ma in pratica spesso ci sono più funzioni correlate inseribili nella stessa interfaccia, quindi la complicazione sintattica per ogni funzione sarà molto inferiore.

Se si è preoccupati per la resa prestazionale sostenuta dalle funzioni virtuali e la profilazione conferma tale preoccupazione, lo si può combinare con la ricetta per i *mock di metodi non-virtual*.

In alternativa, invece di introdurre una nuova interfaccia, si può riscrivere il codice per accettare una std::function invece della funzione libera per poi utilizzare *MockFunction* per mock-are la std::function.

10.1.7 Le Macro `MOCK_METHODn` Old-Style

Prima che la macro generica `MOCK_METHOD` venisse introdotta nel 2018, i mock venivano creati utilizzando una famiglia di macro chiamate `MOCK_METHODn`. Queste macro sono ancora supportate, sebbene sia consigliata la migrazione al nuovo `MOCK_METHOD`.

Le macro nella famiglia `MOCK_METHODn` differiscono da `MOCK_METHOD`:

- La struttura generale è `MOCK_METHODn(MethodName, ReturnType(Args))`, anziché `MOCK_METHOD(ReturnType, MethodName, (Args))`.
- Il numero `n` deve essere uguale al numero di argomenti.
- Quando si crea il mock di un metodo const, è necessario utilizzare `MOCK_CONST_METHODn`.
- Quando si crea il mock di una classe template, il nome della macro deve avere il suffisso `_T`.
- Per specificare il tipo di chiamata, il nome della macro deve avere il suffisso `_WITH_CALLTYPE` e il tipo di chiamata è il primo argomento della macro.

Le vecchie macro e i loro nuovi equivalenti:

10.1.8 Nice, Strict e Naggy {#NiceStrictNaggy}

Se un metodo mock non ha specifiche `EXPECT_CALL` ma viene chiamato, si dice che è una ńnon interessanteż e verrà eseguita lazione di default (specificabile con `ON_CALL()`). Attualmente, una chiamata non interessante farà sì che gMock stampi un warning per default.

Tuttavia, a volte si vorrebbero ignorare queste chiamate poco interessanti, altre volte le si vorrebbe trattare come errori. gMock consente di decidere per ciascun oggetto mock.

Supponiamo che il test utilizzi una classe mock `MockFoo`:

```
TEST(...) {
  MockFoo mock_foo;
  EXPECT_CALL(mock_foo, DoThis());
  ... code that uses mock_foo ...
}
```

Se viene chiamato un metodo di `mock_foo` diverso da `DoThis()`, si otterrà un warning. Tuttavia, se si riscrive il test per utilizzare invece `NiceMock<MockFoo>`, si può sopprimere il warning:

```
using ::testing::NiceMock;

TEST(...) {
  NiceMock<MockFoo> mock_foo;
  EXPECT_CALL(mock_foo, DoThis());
  ... code that uses mock_foo ...
}
```

`NiceMock<MockFoo>` è una sottoclasse di `MockFoo`, quindi può essere utilizzata ovunque sia accettata `MockFoo`.

Funziona anche se il costruttore di `MockFoo` accetta alcuni argomenti, poiché `NiceMock<MockFoo>` ńereditaż i costruttori di `MockFoo`:

```
using ::testing::NiceMock;

TEST(...) {
  NiceMock<MockFoo> mock_foo(5, "hi");  // Calls MockFoo(5, "hi").
  EXPECT_CALL(mock_foo, DoThis());
  ... code that uses mock_foo ...
}
```

Lutilizzo di `StrictMock` è simile, tranne per il fatto che trasforma tutte le chiamate ńnon interessantiż in errori:

```
using ::testing::StrictMock;

TEST(...) {
  StrictMock<MockFoo> mock_foo;
  EXPECT_CALL(mock_foo, DoThis());
  ... code that uses mock_foo ...

  // The test will fail if a method of mock_foo other than DoThis()
  // is called.
}
```

{: .callout .note} NOTA: `NiceMock` e `StrictMock` influenzano solo le chiamate *non interessanti* (chiamate di *metodi* senza [expectation]); non influenzano le chiamate *unexpected* (chiamate di metodi con expectation, ma che non corrispondono). Consultare *Le Chiamate Poco Interessanti e Quelle Unexpected*.

Ci sono però alcuni avvertimenti (purtroppo sono effetti collaterali delle limitazioni del C++):

1. `NiceMock<MockFoo>` e `StrictMock<MockFoo>` funzionano solo per i metodi mock definiti utilizzando la macro `MOCK_METHOD` **direttamente** nella classe `MockFoo`. Se un metodo mock è definito in una **classe**

base di `MockFoo`, il modificatore ńniceż o ństrictż potrebbe non influenzarlo, a seconda del compilatore. In particolare, la nidificazione di `NiceMock` e `StrictMock` (ad esempio `NiceMock<StrictMock<MockFoo>>`) **non** è supportata.

2. `NiceMock<MockFoo>` e `StrictMock<MockFoo>` potrebbero non funzionare correttamente se il distruttore di `MockFoo` non è virtuale. Vorremmo risolvere questo problema, ma è necessario ripulire i test esistenti.

Infine, si deve porre **molta attenzione** su quando utilizzare mock ńnaggyż o ństrictż, poiché tendono a rendere i test più fragili e più difficili da mantenere. Quando si esegue il refactoring del codice senza modificarne il comportamento visibile dallesterno, idealmente si dovrebbe aver bisogno di aggiornare alcun test. Se il codice interagisce con un mock ńnaggyż, però, si potrebbe iniziare a ricevere notevoli warning come risultato della modifica. Peggio ancora, se il codice interagisce con un mock ństrictż, i test potrebbero iniziare a fallire e si sarà costretti a correggerli. La nostra raccomandazione generale è quella di utilizzare mock ńniceż (non ancora il default) per la maggior parte del tempo, usare mock ńnaggyż (lattuale default) durante lo sviluppo o il debug dei test e usare mock ństrictż solo come ultima risorsa.

10.1.9 Semplificare lInterfaccia senza Compromettere il Codice Esistente {#SimplerInterfaces}

A volte un metodo ha un lungo elenco di argomenti che nella maggior parte dei casi non sono interessanti. Per esempio:

```
class LogSink {
 public:
  ...
  virtual void send(LogSeverity severity, const char* full_filename,
                    const char* base_filename, int line,
                    const struct tm* tm_time,
                    const char* message, size_t message_len) = 0;
};
```

Lelenco degli argomenti di questo metodo è lungo e difficile da gestire (largomento `message` non è nemmeno ń0-terminatedż). Se si crea il mock così comè, il suo uso risulterà problematico. Se, tuttavia, proviamo a semplificare questa interfaccia, dovremo sistemare tutti i client che dipendono da essa, il che spesso è irrealizzabile.

Il trucco sta nel redistribuire il metodo nella classe mock:

```
class ScopedMockLog : public LogSink {
 public:
  ...
  void send(LogSeverity severity, const char* full_filename,
                    const char* base_filename, int line, const tm* tm_time,
                    const char* message, size_t message_len) override {
    // We are only interested in the log severity, full file name, and
    // log message.
    Log(severity, full_filename, std::string(message, message_len));
  }

  // Implements the mock method:
  //
  //   void Log(LogSeverity severity,
  //            const string& file_path,
  //            const string& message);
  MOCK_METHOD(void, Log,
              (LogSeverity severity, const string& file_path,
               const string& message));
};
```

Definendo un nuovo metodo mock con un elenco di argomenti ridotto, rendiamo la classe mock più user-friendly.

Questa tecnica può essere applicata anche per facilitare la creazione di mock dei metodi [overloaded]. Ad esempio, quando sono stati utilizzati gli overload per implementare argomenti di default:

```
class MockTurtleFactory : public TurtleFactory {
 public:
  Turtle* MakeTurtle(int length, int weight) override { ... }
  Turtle* MakeTurtle(int length, int weight, int speed) override { ... }

  // the above methods delegate to this one:
  MOCK_METHOD(Turtle*, DoMakeTurtle, ());
};
```

Ciò consente di avere test a cui non interessa quale overload sia stato invocato per evitare di specificare i matcher degli argomenti:

```
ON_CALL(factory, DoMakeTurtle)
    .WillByDefault(Return(MakeMockTurtle()));
```

10.1.10 Alternativa ai Mock delle Classi Concrete

Spesso ci si ritrova a utilizzare classi che non implementano interfacce. Per testare il codice che utilizza una classe di questo tipo (chiamiamola `Concrete`), si potrebbe essere tentati di rendere virtuali i metodi di `Concrete` e crearne il mock.

Cercare di non farlo.

Rendere virtuale una funzione non virtuale è una decisione importante. Si crea un punto di estensione in cui le sottoclassi possono modificare il comportamento della classe. Ciò indebolisce il controllo sulla classe perché ora è più difficile mantenere le invarianti della classe. Si deve rendere virtuale una funzione solo quando esiste un motivo valido per cui una sottoclasse la debba sovrascrivere [override].

Creare direttamente mock di classi concrete è problematico in quanto crea uno stretto accoppiamento tra la classe e i test: qualsiasi piccolo cambiamento nella classe può invalidare i test e rendere difficile la manutenzione dei test.

Per evitare tali problemi, molti programmatori praticano la ńcodificazione delle interfacceż: invece di parlare con la classe `Concrete`, il codice dovrebbe definire uninterfaccia e comunicare tramite essa. Quindi si implementa linterfaccia come un adattatore al di sopra di `Concrete`. Nei test, si puoi facilmente creare il mock dellinterfaccia per osservare come sta andando il codice.

Questa tecnica comporta un po di lavoro:

- Si paga il costo delle chiamate di funzioni virtuali (di solito non è un problema).
- Cè più astrazione da imparare per i programmatori.

Tuttavia, si possono anche apportare benefici significativi oltre a una migliore testabilità:

- LAPI di `Concrete` potrebbe non adattarsi molto bene al dominio del problema, poiché si potrebbe non essere lunico cliente che si tenta di servire. Ne progettare linterfaccia, si ha la possibilità di adattarla alle proprie esigenze: si possono aggiungere funzionalità di livello superiore, rinominare elementi, ecc. invece di limitarsi a ritagliare la classe. Ciò consente di scrivere il codice (lutilizzatore dellinterfaccia) in un modo più naturale, il che significa che sarà più leggibile, più gestibile e si sarà più produttivi.
- Se limplementazione di `Concrete` dovesse cambiare, non sarà necessario riscriverla ovunque venga utilizzata. Al contrario, si possono assorbire le modifiche nellimplementazione dellinterfaccia e laltro codice e gli altri test saranno isolati da queste modifiche.

Qualcuno teme che se tutti praticassero questa tecnica, si finirebbe per scrivere molto codice ridondante. Questa preoccupazione è del tutto comprensibile. Tuttavia, ci sono due ragioni per cui potrebbe non essere così:

- Progetti diversi potrebbero dover utilizzare `Concrete` in modi diversi, quindi le interfacce migliori per loro saranno diverse. Pertanto, ognuno di essi avrà la propria interfaccia specifica del dominio al di sopra di `Concrete`, e non avranno lo stesso codice.

- Se un numero sufficiente di progetti vuole utilizzare la stessa interfaccia, può sempre condividerla, proprio come hanno condiviso `Concrete`. Si può archiviare linterfaccia e ladattatore [adaptor] da qualche parte vicino a `Concrete` (probabilmente in una sotto-directory `contrib`) e lasciare che molti progetti la utilizzino.

Si devono valutare attentamente i pro e i contro per il problema particolare, ma certamente la comunità Java lo pratica da molto tempo ed è una tecnica comprovata ed efficace applicabile in unampia varietà di situazioni. :-)

10.1.11 Delegare le Chiamate ad una Fake {#DelegatingToFake}

Talvolta si hanno implementazioni [fake] non banali di uninterfaccia. Per esempio:

```
class Foo {
 public:
  virtual ~Foo() {}
  virtual char DoThis(int n) = 0;
  virtual void DoThat(const char* s, int* p) = 0;
};

class FakeFoo : public Foo {
 public:
  char DoThis(int n) override {
    return (n > 0) ? '+' :
           (n < 0) ? '-' : '0';
  }

  void DoThat(const char* s, int* p) override {
    *p = strlen(s);
  }
};
```

Ora si vuole creare la mock di questa interfaccia in modo da potervi impostare delle expectation. Tuttavia, si vuole usare anche `FakeFoo` per il comportamento di default, poiché duplicarlo nelloggetto mock richiede molto lavoro.

Quando si definisce la classe mock utilizzando gMock, si può far sì che deleghi la sua azione di default a una classe fake che già si possiede, utilizzando questo pattern:

```
class MockFoo : public Foo {
 public:
  // Normal mock method definitions using gMock.
  MOCK_METHOD(char, DoThis, (int n), (override));
  MOCK_METHOD(void, DoThat, (const char* s, int* p), (override));

  // Delegates the default actions of the methods to a FakeFoo object.
  // This must be called *before* the custom ON_CALL() statements.
  void DelegateToFake() {
    ON_CALL(*this, DoThis).WillByDefault([this](int n) {
      return fake_.DoThis(n);
    });
    ON_CALL(*this, DoThat).WillByDefault([this](const char* s, int* p) {
      fake_.DoThat(s, p);
    });
  }

 private:
  FakeFoo fake_;  // Keeps an instance of the fake in the mock.
};
```

Con questo si può usare `MockFoo` nei test come al solito. Ricordarsi solo che se non si imposta esplicitamente unazione in una `ON_CALL()` o in una `EXPECT_CALL()`, la fake verrà chiamato a farlo.:

```
using ::testing::_;

TEST(AbcTest, Xyz) {
  MockFoo foo;

  foo.DelegateToFake();  // Enables the fake for delegation.

  // Put your ON_CALL(foo, ...)s here, if any.

  // No action specified, meaning to use the default action.
  EXPECT_CALL(foo, DoThis(5));
  EXPECT_CALL(foo, DoThat(_, _));

  int n = 0;
  EXPECT_EQ(foo.DoThis(5), '+');  // FakeFoo::DoThis() is invoked.
  foo.DoThat("Hi", &n);  // FakeFoo::DoThat() is invoked.
  EXPECT_EQ(n, 2);
}
```

Alcuni suggerimenti:

- Volendo, si può comunque sovrascrivere lazione di default fornendo la propria `ON_CALL()` o utilizzando `.WillOnce()` / `.WillRepeatedly()` in `EXPECT_CALL()`.
- In `DelegateToFake()`, si devono solo delegare i metodi di cui si intende utilizzare limplementazione fake.
- La tecnica generale discussa qui funziona per i metodi [overloaded], ma si dovrà dire al compilatore quale versione si intende. Per chiarire le ambiguità di una funzione mock (quella specificata tra parentesi di `ON_CALL()`), utilizzare *questa tecnica*; per disambiguare una funzione fake (quella che si inserisce allinterno di `Invoke()`), usare uno `static_cast` per specificare il tipo della funzione. Ad esempio, se la classe `Foo` ha i metodi `char DoThis(int n)` e `bool DoThis(double x) const`, per invocare questultimo, si deve scrivere `Invoke(&fake_, static_cast<bool (FakeFoo::*)(double) const>(&FakeFoo::DoThis))` invece di `Invoke(&fake_, &FakeFoo::DoThis)` (La cosa strana allinterno delle parentesi angolari di `static_cast` è il tipo di puntatore a funzione al secondo metodo `DoThis()`).
- Dover mischiare una mock con una fake è spesso un segno che qualcosa è andato storto. Forse non si è ancora abituati al metodo di test basato sullinterazione [interaction-based]. Oppure probabilmente linterfaccia sta assumendo troppi ruoli e dovrebbe essere suddivisa. Pertanto, **non abusarne**. Consigliamo di farlo solo come passaggio intermedio nelleffettuare il refactoring del codice.

Per quanto riguarda il suggerimento su come mescolare un mock e un fake, ecco un esempio del perché potrebbe essere un brutto segno: supponiamo di avere una classe `System` per operazioni di sistema di basso livello. In particolare, vengono eseguite operazioni su file e I/O. E supponiamo che si voglia testare come il codice utilizza `System` per eseguire I/O e si voglia semplicemente che le operazioni sui file funzionino normalmente. Se si crea la mock di tutta la classe `System`, si dovrà fornire unimplementazione fake per la parte relativa alle operazioni sui file, il che suggerisce che `System` sta assumendo troppi ruoli.

Si può invece definire uninterfaccia `FileOps e un'interfaccia IOOpsdividendo le funzionalità diSystemin due. Poi si può creare la mock diIOOpssenza avere un mock diFileOps`.

10.1.12 Delegare le Chiamate allOggetto Reale

Quando si utilizzano copie di test (mock, fake, stub e così via), a volte i loro comportamenti differiranno da quelli degli oggetti reali. Questa differenza potrebbe essere intenzionale (come nella simulazione di un errore in modo da poter testare il codice di gestione degli errori) o involontaria. Se per errore i mock hanno comportamenti diversi rispetto agli oggetti reali, ci si potrebbe ritrovare con un codice che supera i test ma fallisce in produzione.

Si può utilizzare la tecnica *delegating-to-real* per fare in modo che il mock abbia lo stesso comportamento delloggetto reale pur mantenendo la capacità di convalidare le chiamate. Questa tecnica è molto simile alla *delegating-to-fake*, con la differenza che utilizziamo un oggetto reale invece di un fake. Ecco un esempio:

```
using ::testing::AtLeast;

class MockFoo : public Foo {
 public:
  MockFoo() {
    // By default, all calls are delegated to the real object.
    ON_CALL(*this, DoThis).WillByDefault([this](int n) {
      return real_.DoThis(n);
    });
    ON_CALL(*this, DoThat).WillByDefault([this](const char* s, int* p) {
      real_.DoThat(s, p);
    });
    ...
  }
  MOCK_METHOD(char, DoThis, ...);
  MOCK_METHOD(void, DoThat, ...);
  ...
 private:
  Foo real_;
};

...
  MockFoo mock;
  EXPECT_CALL(mock, DoThis())
      .Times(3);
  EXPECT_CALL(mock, DoThat("Hi"))
      .Times(AtLeast(1));
  ... use mock in test ...
```

Con questo, gMock verificherà che il codice abbia effettuato le chiamate giuste (con gli argomenti giusti, nell'ordine giusto, chiamato il numero giusto di volte, ecc.) e un oggetto reale risponderà alle chiamate (quindi il comportamento sarà lo stesso come in produzione). Questo fornisce il meglio di entrambi i mondi.

10.1.13 Delegare le Chiamate a una Classe Genitore

Idealmente, si dovrebbero codificare le interfacce, i cui metodi sono tutti puramente virtuali. In realtà, a volte è necessario creare la mock di un metodo virtuale che non è puro (ovvero ha già un'implementazione). Per esempio:

```
class Foo {
 public:
  virtual ~Foo();

  virtual void Pure(int n) = 0;
  virtual int Concrete(const char* str) { ... }
};

class MockFoo : public Foo {
 public:
  // Mocking a pure method.
  MOCK_METHOD(void, Pure, (int n), (override));
  // Mocking a concrete method.  Foo::Concrete() is shadowed.
  MOCK_METHOD(int, Concrete, (const char* str), (override));
};
```

A volte si vorrebbe chiamare `Foo::Concrete()` invece di `MockFoo::Concrete()`. Probabilmente come parte di un'azione stub, o forse il test non ha bisogno del mock di `Concrete()` (ma sarebbe davvero tedioso dover definire una nuova classe mock ogni volta che non è necessario il mock di uno dei suoi metodi).

Si può chiamare `Foo::Concrete()` allinterno di unazione:

```
...
  EXPECT_CALL(foo, Concrete).WillOnce([&foo](const char* str) {
    return foo.Foo::Concrete(str);
  });
```

oppure dire al mock delloggetto che non si vuole il mock di `Concrete()`:

```
...
  ON_CALL(foo, Concrete).WillByDefault([&foo](const char* str) {
    return foo.Foo::Concrete(str);
  });
```

(Perché non scrivere semplicemente `{ return foo.Concrete(str); }`? Se lo si fa, verrà chiamato `MockFoo::Concrete()` (provocando una ricorsione infinita) in quanto `Foo::Concrete()` è virtuale. È così che funziona il C++).

10.2 Uso dei Matcher

10.2.1 Matching Esatto dei Valori degli Argomenti

Si può specificare esattamente quali argomenti si aspetta un metodo mock:

```
using ::testing::Return;
...
  EXPECT_CALL(foo, DoThis(5))
      .WillOnce(Return('a'));
  EXPECT_CALL(foo, DoThat("Hello", bar));
```

10.2.2 Uso di Semplici Matcher

I matcher si possono usare per abbinare [match] argomenti che hanno una certa proprietà:

```
using ::testing::NotNull;
using ::testing::Return;
...
  EXPECT_CALL(foo, DoThis(Ge(5)))  // The argument must be >= 5.
      .WillOnce(Return('a'));
  EXPECT_CALL(foo, DoThat("Hello", NotNull()));
      // The second argument must not be NULL.
```

Un matcher usato spesso è _, che corrisponde a qualsiasi cosa:

```
  EXPECT_CALL(foo, DoThat(_, NotNull()));
```

10.2.3 Combinazione di Matcher {#CombiningMatchers}

Si possono creare abbinamenti complessi da quelli esistenti utilizzando `AllOf()`, `AllOfArray()`, `AnyOf()`, `AnyOfArray()` e `Not()`:

```
using ::testing::AllOf;
using ::testing::Gt;
using ::testing::HasSubstr;
using ::testing::Ne;
using ::testing::Not;
...
```

(continues on next page)

(continua dalla pagina precedente)

```
  // The argument must be > 5 and != 10.
  EXPECT_CALL(foo, DoThis(AllOf(Gt(5),
                                Ne(10))));

  // The first argument must not contain sub-string "blah".
  EXPECT_CALL(foo, DoThat(Not(HasSubstr("blah")),
                          NULL));
```

I matcher sono oggetti funzione e quelli parametrizzati possono essere composti proprio come qualsiasi altra funzione. However because their types can be long and rarely provide meaningful information, it can be easier to express them with template parameters and `auto`. Per esempio,

```
using ::testing::Contains;
using ::testing::Property;

template <typename SubMatcher>
inline constexpr auto HasFoo(const SubMatcher& sub_matcher) {
  return Property("foo", &MyClass::foo, Contains(sub_matcher));
};
...
  EXPECT_THAT(x, HasFoo("blah"));
```

10.2.4 Casting dei Matcher {#SafeMatcherCast}

I matcher di gMock sono tipizzati staticamente, il che significa che il compilatore può individuare lerrore se si usa un matcher del tipo sbagliato (ad esempio, se si usa `Eq(5)` per trovare una corrispondenza con un argomento `string`). Una cosa buona!

A volte, però, sapendo cosa si fa, si vuole che il compilatore dia un po di tregua. Un esempio è che si abbia un matcher per `long` e largomento che si vuol trovare è `int`. Sebbene i due tipi non siano esattamente gli stessi, non cè niente di veramente sbagliato nellusare un `Matcher<long>` per abbinare un `int` - dopo tutto, possiamo convertire prima largomento `int` in un `long` senza perdere informazioni prima di passarlo al matcher.

Per supportare questa esigenza, gMock offre la funzione `SafeMatcherCast<T>(m)`. Si effettua il cast di un matcher `m` nel tipo `Matcher<T>`. Per garantire la sicurezza, gMock verifica che sia `U` il tipo accettato da `m`:

1. Il tipo `T` può essere convertito [cast] *implicitamente* nel tipo `U`;
2. Quando sia `T` che `U` sono tipi aritmetici nativi (`bool`, integer e numeri in virgola mobile), la conversione da `T` a `U` è senza perdita di informazioni (in altre parole, qualsiasi valore rappresentabile da `T` può essere rappresentato da `U`); e
3. When `U` is a non-const reference, `T` must also be a reference (as the underlying matcher may be interested in the address of the `U` value).

Il codice non verrà compilato se una di queste condizioni non viene soddisfatta.

Ecco un esempio:

```
using ::testing::SafeMatcherCast;

// A base class and a child class.
class Base { ... };
class Derived : public Base { ... };

class MockFoo : public Foo {
 public:
  MOCK_METHOD(void, DoThis, (Derived* derived), (override));
};
```

(continues on next page)

(continua dalla pagina precedente)

```
...
  MockFoo foo;
  // m is a Matcher<Base*> we got from somewhere.
  EXPECT_CALL(foo, DoThis(SafeMatcherCast<Derived*>(m)));
```

Se si ritiene la `SafeMatcherCast<T>(m)` troppo limitante, si può usare una funzione simile `MatcherCast<T>(m)`. La differenza è che `MatcherCast` funziona fin quando si può eseguire lo `static_cast` del tipo `T` nel tipo `U`.

`MatcherCast` consente essenzialmente di aggirare il sistema di tipi di C++ (`static_cast` non è sempre sicuro in quanto potrebbe eliminare informazioni, ad esempio), si faccia quindi attenzione a non abusarne.

10.2.5 Scegliere le Funzioni Overloaded {#SelectOverload}

Se ci si aspetta che venga chiamata una funzione overloaded, il compilatore potrebbe aver bisogno di aiuto sulla scelta della versione.

Per chiarire le ambiguità delle funzioni overloaded tramite lessere const di questo oggetto, si usa largomento wrapper di `Const()`.

```
using ::testing::ReturnRef;

class MockFoo : public Foo {
  ...
  MOCK_METHOD(Bar&, GetBar, (), (override));
  MOCK_METHOD(const Bar&, GetBar, (), (const, override));
};

...
  MockFoo foo;
  Bar bar1, bar2;
  EXPECT_CALL(foo, GetBar())         // The non-const GetBar().
      .WillOnce(ReturnRef(bar1));
  EXPECT_CALL(Const(foo), GetBar())  // The const GetBar().
      .WillOnce(ReturnRef(bar2));
```

(`Const()` è definito da gMock e restituisce un riferimento `const` al suo argomento).

Per chiarire le ambiguità delle funzioni overloaded con lo stesso numero di argomenti ma tipi di argomento diversi, potrebbe essere necessario specificare il tipo esatto di un matcher, racchiudendolo in `Matcher<type>()` o utilizzando un matcher il cui tipo è fisso (`TypedEq<type>`, `An<type>()`, ecc.):

```
using ::testing::An;
using ::testing::Matcher;
using ::testing::TypedEq;

class MockPrinter : public Printer {
 public:
  MOCK_METHOD(void, Print, (int n), (override));
  MOCK_METHOD(void, Print, (char c), (override));
};

TEST(PrinterTest, Print) {
  MockPrinter printer;

  EXPECT_CALL(printer, Print(An<int>()));            // void Print(int);
  EXPECT_CALL(printer, Print(Matcher<int>(Lt(5))));  // void Print(int);
  EXPECT_CALL(printer, Print(TypedEq<char>('a')));   // void Print(char);
```

(continues on next page)

(continua dalla pagina precedente)

```
  printer.Print(3);
  printer.Print(6);
  printer.Print('a');
}
```

10.2.6 Eseguire Azioni Diverse in Base agli Argomenti

Quando viene chiamato un metodo mock, verrà selezionato l*ultimo* matching expectation ancora attiva (si pensi che ńil più recente sovrascrive il più vecchioż). Quindi, si può fare in modo che un metodo faccia cose diverse a seconda dei valori dei suoi argomenti in questo modo:

```
using ::testing::_;
using ::testing::Lt;
using ::testing::Return;
...
  // The default case.
  EXPECT_CALL(foo, DoThis(_))
      .WillRepeatedly(Return('b'));
  // The more specific case.
  EXPECT_CALL(foo, DoThis(Lt(5)))
      .WillRepeatedly(Return('a'));
```

Ora, se `foo.DoThis()` viene chiamato con un valore inferiore a 5, verrà restituito `'a'`; altrimenti verrà restituito `'b'`.

10.2.7 Corrispondenza di Più Argomenti come un Tuttuno

A volte non è sufficiente abbinare [match] gli argomenti individualmente. Ad esempio, potremmo voler dire che il primo argomento deve essere minore del secondo argomento. La clausola `With()` ci consente di abbinare tutti gli argomenti di una funzione mock nel suo insieme. Per esempio,

```
using ::testing::_;
using ::testing::Ne;
using ::testing::Lt;
...
  EXPECT_CALL(foo, InRange(Ne(0), _))
      .With(Lt());
```

dice che il primo argomento di `InRange()` non deve essere 0 e deve essere minore del secondo argomento.

Lespressione in `With()` devessere un matcher del tipo `Matcher<std::tuple<A1, ..., An»`, dove `A1`, , `An` sono i tipi degli argomenti della funzione.

Si può anche scrivere `AllArgs(m)` invece di `m` in `.With()`. Le due forme sono equivalenti, ma `.With(AllArgs(Lt()))` è più leggibile di `.With(Lt())`.

Si può usare `Args<k1, ..., kn>(m)` per far corrispondere gli `n` argomenti selezionati (come una tupla) con `m`. Per esempio,

```
using ::testing::_;
using ::testing::AllOf;
using ::testing::Args;
using ::testing::Lt;
...
  EXPECT_CALL(foo, Blah)
      .With(AllOf(Args<0, 1>(Lt()), Args<1, 2>(Lt())));
```

dice che `Blah` verrà chiamato con gli argomenti `x`, `y`, e `z` dove `x < y < z`. Si noti che in questo esempio, non è stato necessario specificare i matcher posizionali.

Per comodità ed esempio, gMock fornisce alcuni matcher per 2-tuple, incluso il matcher `Lt()` precedente. Consultare *Multi-argument Matchers* per la lista completa.

Si noti che per passare gli argomenti a un proprio predicato (p.es. `.With(Args<0, 1>(Truly(&MyPredicate)))`), quel predicato DEVE essere scritto per accettare un `std::tuple` come argomento; gMock passerà gli `n` argomenti selezionati come *una* singola tupla al predicato.

10.2.8 Usare i Matcher come Predicati

Si è notato che un matcher è solo un fantasioso predicato che sa anche come descrivere se stesso? Molti algoritmi esistenti accettano predicati come argomenti (ad esempio quelli definiti nellheader <`algorithm`> di STL) e sarebbe un peccato se ai matcher gMock non fosse consentito partecipare.

Fortunatamente, si può utilizzare un matcher in cui è previsto un funtore del predicato unario racchiudendolo nella funzione `Matches()`. Per esempio,

```
#include <algorithm>
#include <vector>

using ::testing::Matches;
using ::testing::Ge;

vector<int> v;
...
// How many elements in v are >= 10?
const int count = count_if(v.begin(), v.end(), Matches(Ge(10)));
```

Dal momento che si possono creare facilmente matcher complessi da quelli più semplici utilizzando gMock, questo fornisce un modo per costruire comodamente predicati compositi (fare lo stesso usando lheader <`functional`> di STL è semplicemente complicato). Per esempio, ecco un predicato soddisfatto da qualsiasi numero >= 0, <= 100 e != 50:

```
using ::testing::AllOf;
using ::testing::Ge;
using ::testing::Le;
using ::testing::Matches;
using ::testing::Ne;
...
Matches(AllOf(Ge(0), Le(100), Ne(50)))
```

10.2.9 Uso dei Matcher nelle Asserzioni di googletest

Vedere *EXPECT_THAT* nelle ńAsserzioni di Eccezioniż.

10.2.10 Uso dei Predicati come Matcher

gMock fornisce un insieme di matcher nativi per far corrispondere gli argomenti con i valori attesi: per ulteriori informazioni consultare i *Riferimenti ai Matcher*. Nel caso in cui si trovi carente il set nativo, si può utilizzare una funzione di predicato unario arbitrario o un funtore come matcher, purché il predicato accetti un valore del tipo desiderato. Lo si può fare racchiudendo il predicato allinterno della funzione `Truly()`, per esempio:

```
using ::testing::Truly;

int IsEven(int n) { return (n % 2) == 0 ? 1 : 0; }
...
  // Bar() must be called with an even number.
  EXPECT_CALL(foo, Bar(Truly(IsEven)));
```

Notare che la funzione/funtore del predicato non deve restituire `bool`. Funziona fin quando il valore restituito può essere utilizzato come condizione nellistruzione `if (condition) ...`.

10.2.11 Match di Argomenti che Non Sono Copiabili

Quando si esegue un `EXPECT_CALL(mock_obj, Foo(bar))`, gMock salva una copia di `bar`. Quando `Foo()` viene chiamato in seguito, gMock confronta largomento di `Foo()` con la copia salvata di `bar`. In questo modo, non ci si deve preoccupare che la `bar` venga modificata o distrutta dopo lesecuzione di `EXPECT_CALL()`. Lo stesso vale quando si usano matcher come `Eq(bar)`, `Le(bar)` e così via.

Ma cosa succede se `bar` non può essere copiato (cioè non ha un costruttore di copia)? Si potrebbe definire la propria funzione matcher o callback e usarla con `Truly()`, come mostrato nelle due ricette precedenti. Oppure si potrebbe riuscire a evitarlo se se si garantisce che la `bar` non verrà modificata dopo lesecuzione di `EXPECT_CALL()`. Basta dire a gMock che dovrebbe salvare un riferimento a `bar`, invece di una sua copia. Ecco come:

```
using ::testing::Eq;
using ::testing::Lt;
...
  // Expects that Foo()'s argument == bar.
  EXPECT_CALL(mock_obj, Foo(Eq(std::ref(bar))));

  // Expects that Foo()'s argument < bar.
  EXPECT_CALL(mock_obj, Foo(Lt(std::ref(bar))));
```

Da ricordare: se lo si fa, non modificare `bar` dopo la `EXPECT_CALL()`, altrimenti il risultato non sarà definito.

10.2.12 Validare un Membro di un Oggetto

Spesso una funzione mock accetta un riferimento alloggetto come argomento. Quando si esegue il matching di argomenti, si potrebbe non voler confrontare lintero oggetto con un oggetto fisso, poiché ciò potrebbe essere eccessivo. Potrebbe invece essere necessario validare una determinata variabile membro o il risultato di un determinato metodo getter delloggetto. Lo si può fare con `Field()` e con `Property()`. Più specificamente,

```
Field(&Foo::bar, m)
```

è un matcher che corrisponde a un oggetto `Foo` la cui variabile membro `bar` soddisfa il matcher `m`.

```
Property(&Foo::baz, m)
```

è un matcher che corrisponde a un oggetto `Foo` il cui metodo `baz()` restituisce un valore che soddisfa il matcher `m`.

Per esempio:

Espressione	Descrizione
`Field(&Foo::number, Ge(3))`	Corrisponde a `x` dove `x.number >= 3`.
`Property(&Foo::name, StartsWith("John "))`	Corrisponde a `x` dove `x.name()` inizia con `"John "`.

Notare che in `Property(&Foo::baz, ...)`, il metodo `baz()` non deve accettare argomenti ed essere dichiarato come `const`. Non usare `Property()` contro funzioni membro che non si possiedono, perché prendere gli indirizzi delle funzioni è ńfragileż e generalmente non fa parte del contratto della funzione.

`Field()` e `Property()` possono anche eseguire il match di semplici puntatori a oggetti. Per esempio,

```
using ::testing::Field;
using ::testing::Ge;
...
Field(&Foo::number, Ge(3))
```

corrisponde a un semplice puntatore `p` dove `p->number >= 3`. Se `p` è `NULL`, il match fallirà sempre indipendentemente dal matcher interno.

Cosa succede se si validano più di un membro contemporaneamente? Si ricorda che ci sono *`AllOf()` e `AllOfArray()`*.

Infine `Field()` e `Property()` forniscono gli overload che accettano i nomi del campo o della proprietà come primo argomento per includerlo nel messaggio di errore. Questo può essere utile quando si creano matcher combinati.

```
using ::testing::AllOf;
using ::testing::Field;
using ::testing::Matcher;
using ::testing::SafeMatcherCast;

Matcher<Foo> IsFoo(const Foo& foo) {
  return AllOf(Field("some_field", &Foo::some_field, foo.some_field),
               Field("other_field", &Foo::other_field, foo.other_field),
               Field("last_field", &Foo::last_field, foo.last_field));
}
```

10.2.13 Validare il Valore Puntato da un Argomento Puntatore

Le funzioni C++ spesso accettano puntatori come argomenti. Si possono usare matcher come `IsNull()`, `NotNull()` e altri per avere una corrispondenza con un puntatore, ma cosa succede se ci si vuol accertare che il valore *puntato* dal puntatore, anziché il puntatore stesso, ha una certa proprietà? Ebbene, si può usare il matcher `Pointee(m)`.

`Pointee(m)` esegue il matching di un puntatore se e solo se `m` corrisponde al valore a cui punta il puntatore. Per esempio:

```
using ::testing::Ge;
using ::testing::Pointee;
...
  EXPECT_CALL(foo, Bar(Pointee(Ge(3))));
```

si aspetta che `foo.Bar()` venga chiamato con un puntatore che punta a un valore maggiore o uguale a 3.

Una cosa bella di `Pointee()` è che tratta un puntatore `NULL` come una corrispondenza fallita, quindi si può scrivere `Pointee(m)` anziché

```
using ::testing::AllOf;
using ::testing::NotNull;
using ::testing::Pointee;
...
  AllOf(NotNull(), Pointee(m))
```

senza preoccuparsi che un puntatore `NULL` mandi in crash il test.

Inoltre, abbiamo detto che `Pointee()` funziona sia con puntatori [raw] **sia con** puntatori smart (`std::unique_ptr`, `std::shared_ptr`, ecc.)?

Cosa succede se si ha un puntatore a puntatore? Indovinato - si può usare `Pointee()` nidificato scavare più a fondo il valore. Per esempio, `Pointee(Pointee(Lt(3)))` corrisponde a un puntatore che punta a un puntatore che punta a un numero inferiore a 3 (che salti).

10.2.14 Definizione di una Classe Matcher Custom {#CustomMatcherClass}

La maggior parte dei matcher può essere definita semplicemente utilizzando *le macro MATCHER**, che sono concise e flessibili e producono buoni messaggi di errore. Tuttavia, queste macro non sono molto esplicite riguardo alle interfacce che creano e non sono sempre adatte, soprattutto per i matcher che verranno ampiamente riutilizzati.

Per i casi più avanzati, potrebbe essere necessario definire la propria classe matcher. Un matcher custom consente di testare una proprietà invariante specifica di quelloggetto. Vediamo come farlo.

Immaginiamo di avere una funzione mock che accetta un oggetto di tipo `Foo`, che ha un metodo `int bar()` e un metodo `int baz()`. Si vuol vincolare il fatto che il valore dellargomento di `bar()` più il valore del suo `baz()` sia un certo numero. (Questo è un invariante). Ecco come possiamo scrivere e utilizzare una classe matcher per farlo:

```
class BarPlusBazEqMatcher {
 public:
  using is_gtest_matcher = void;

  explicit BarPlusBazEqMatcher(int expected_sum)
      : expected_sum_(expected_sum) {}

  bool MatchAndExplain(const Foo& foo,
                       std::ostream* /* listener */) const {
    return (foo.bar() + foo.baz()) == expected_sum_;
  }

  void DescribeTo(std::ostream* os) const {
    *os << "bar() + baz() equals " << expected_sum_;
  }

  void DescribeNegationTo(std::ostream* os) const {
    *os << "bar() + baz() does not equal " << expected_sum_;
  }
 private:
  const int expected_sum_;
};

::testing::Matcher<const Foo&> BarPlusBazEq(int expected_sum) {
  return BarPlusBazEqMatcher(expected_sum);
}

...
  Foo foo;
  EXPECT_THAT(foo, BarPlusBazEq(5))...;
```

10.2.15 Il Matching dei Contenitori

A volte un container STL (ad esempio lista, vettore, mappa,) viene passato a una funzione mock e lo si vorrebbe convalidare. Poiché la maggior parte dei container STL supportano loperatore ==, si può scrivere `Eq(expected_container)` o semplicemente `expected_container` per verificare esattamente un container.

A volte, però, si necessita di una maggiore flessibilità (ad esempio, il primo elemento deve corrispondere esattamente, ma il secondo elemento può essere qualsiasi numero positivo e così via). Inoltre, i container utilizzati nei test hanno spesso un numero limitato di elementi e doverli definire [out-of-line] è un po complicato.

In questi casi si può utilizzare il matcher `ElementsAre()` o `UnorderedElementsAre()`:

```
using ::testing::_;
using ::testing::ElementsAre;
using ::testing::Gt;
...
  MOCK_METHOD(void, Foo, (const vector<int>& numbers), (override));
...
  EXPECT_CALL(mock, Foo(ElementsAre(1, Gt(0), _, 5)));
```

Il matcher sopra dice che il container deve avere 4 elementi, che devono essere rispettivamente 1, maggiore di 0, qualsiasi cosa e 5.

Se invece si scrive:

```
using ::testing::_;
using ::testing::Gt;
using ::testing::UnorderedElementsAre;
...
  MOCK_METHOD(void, Foo, (const vector<int>& numbers), (override));
...
  EXPECT_CALL(mock, Foo(UnorderedElementsAre(1, Gt(0), _, 5)));
```

Significa che il container deve avere 4 elementi, che (con qualche permutazione) devono essere rispettivamente 1, maggiore di 0, qualsiasi cosa e 5.

In alternativa si possono inserire gli argomenti in un array in stile C e utilizzare invece `ElementsAreArray()` o `UnorderedElementsAreArray()`:

```
using ::testing::ElementsAreArray;
...
  // ElementsAreArray accepts an array of element values.
  const int expected_vector1[] = {1, 5, 2, 4, ...};
  EXPECT_CALL(mock, Foo(ElementsAreArray(expected_vector1)));

  // Or, an array of element matchers.
  Matcher<int> expected_vector2[] = {1, Gt(2), _, 3, ...};
  EXPECT_CALL(mock, Foo(ElementsAreArray(expected_vector2)));
```

Nel caso in cui larray debba essere creato dinamicamente (e quindi la dimensione dellarray non possa essere dedotta dal compilatore), si può fornire a `ElementsAreArray()` un argomento aggiuntivo per specificare la dimensione dellarray:

```
using ::testing::ElementsAreArray;
...
  int* const expected_vector3 = new int[count];
  ... fill expected_vector3 with values ...
  EXPECT_CALL(mock, Foo(ElementsAreArray(expected_vector3, count)));
```

Si usa `Pair` quando si confrontano mappe o altri container associativi.

{% raw %}

```
using ::testing::UnorderedElementsAre;
using ::testing::Pair;
...
  absl::flat_hash_map<string, int> m = {{"a", 1}, {"b", 2}, {"c", 3}};
  EXPECT_THAT(m, UnorderedElementsAre(
      Pair("a", 1), Pair("b", 2), Pair("c", 3)));
```

{% endraw %}

Suggerimenti:

- `ElementsAre*()` è utilizzabile per [to match] *qualsiasi* container che implementi il pattern delliteratore STL (ovvero ha un tipo `const_iterator` e supporta `begin()/end()`), non solo quelli definiti in STL. Funzionerà anche con i tipi di container ancora da scrivere, purché seguano lo schema sopra riportato.
- Si possono usare `ElementsAre*()` nidificati per [to match] container nidificati (multi-dimensionali).
- Se il container viene passato tramite puntatore anziché per riferimento, si scrive semplicemente `Pointee(ElementsAre*(...))`.
- Lordine degli elementi *è importante* per `ElementsAre*()`. Se lo si sta utilizzando con contenitori il cui ordine degli elementi non è definito (come `std::unordered_map`) si deve utilizzare `UnorderedElementsAre`.

10.2.16 Matcher Condivisi

Internamente, un oggetto matcher gMock è costituito da un puntatore a un oggetto di implementazione con conteggio dei riferimenti. La copia dei matcher è consentita ed è molto efficiente, poiché viene copiato solo il puntatore. Quando lultimo matcher che fa riferimento alloggetto di implementazione muore, loggetto di implementazione verrà eliminato.

Pertanto, se si ha un matcher complesso che si vuole utilizzare più e più volte, non è necessario crearlo ogni volta. Basta assegnarlo a una variabile matcher e utilizzare quella variabile ripetutamente! Per esempio,

```
using ::testing::AllOf;
using ::testing::Gt;
using ::testing::Le;
using ::testing::Matcher;
...
  Matcher<int> in_range = AllOf(Gt(5), Le(10));
  ... use in_range as a matcher in multiple EXPECT_CALLs ...
```

10.2.17 I matcher non devono avere effetti collaterali {#PureMatchers}

{: .callout .warning} ATTENZIONE: gMock non garantisce quando o quante volte un matcher verrà invocato. Pertanto, tutti i matcher devono essere *puramente funzionali*: non possono avere effetti collaterali e il risultato della corrispondenza non deve dipendere da nientaltro che dai parametri del matcher e dal valore da abbinare.

Questo requisito deve essere soddisfatto indipendentemente da come viene definito un matcher (ad esempio, se è uno dei matcher standard o un matcher personalizzato). In particolare, un matcher non può mai chiamare una funzione mock, poiché ciò influenzerà lo stato delloggetto mock e di gMock.

10.3 Impostare le Expectation

10.3.1 Quando usare Expect {#UseOnCall}

ON_CALL è probabilmente il *costrutto singolo più sotto-utilizzato* in gMock.

Esistono fondamentalmente due costrutti per definire il comportamento di un oggetto mock: `ON_CALL` and `EXPECT_CALL`. La differenza? `ON_CALL` definisce cosa succede quando viene chiamato un metodo mock, ma non implica alcuna aspettativa [expectation] sul metodo che viene chiamato. `EXPECT_CALL` non solo definisce il comportamento, ma imposta anche laspettativa [expectation] che il metodo venga chiamato con gli argomenti dati, per il numero dato di volte (e *nellordine indicato* quando si specifica anche lordine).

Dato che `EXPECT_CALL` fa di più, non è meglio di `ON_CALL`? Non proprio. Ogni `EXPECT_CALL` aggiunge un vincolo sul comportamento del codice sotto test. Avere più vincoli del necessario è *una pessima idea* - anche peggio che non avere abbastanza vincoli.

Questo potrebbe essere controintuitivo. Come potrebbero i test che verificano di più essere peggiori dei test che verificano di meno? La verifica non è il punto centrale dei test?

La risposta sta in *cosa* dovrebbe verificare un test. **Un buon test verifica il contratto del codice**. Se un test specifica eccessivamente, non lascia abbastanza libertà allimplementazione. Di conseguenza, modificare limplementazione senza infrangere il contratto (ad esempio refactoring e ottimizzazione), cosa che dovrebbe essere perfettamente consentita, può ńrompereż tali test. Quindi si deve dedicare del tempo a risolverli, solo per vederli rotti nuovamente la prossima volta che limplementazione viene modificata.

Si tenga presente che non è necessario verificare più di una proprietà in un test. In effetti, **è buona norma stilistica verificare solo una cosa in un test**. Se lo si fa, un bug probabilmente romperà solo uno o due test invece di dozzine (nel qual caso si vorrebbe eseguire il debug?). Se si ha anche labitudine di dare ai test nomi descrittivi che indichino ciò che verificano, spesso si può facilmente indovinare cosa cè che non va solo dal log.

Quindi usare `ON_CALL` per default e usare `EXPECT_CALL` solo quando si intende effettivamente verificare che la chiamata sia stata effettuata. Ad esempio, si potrebbero avere un sacco di `ON_CALL` nella fixture di test per impostare il comportamento comune del mock condiviso da tutti i test nello stesso gruppo e scrivere (appena) diversi `EXPECT_CALL` in diversi `TEST_F` per verificare diversi aspetti del comportamento del codice. Rispetto allo stile

in cui ogni TEST ha molte EXPECT_CALL, questo porta a test più resilienti alle modifiche implementative (e quindi con meno probabilità di richiedere manutenzione) e rende lintento dei test più ovvi (quindi sono più facili da manutenere).

Se infastidisce dal messaggio ńUninteresting mock function callż stampato quando viene chiamato un metodo mock senza una EXPECT_CALL, si può usare invece un NiceMock per sopprimere tutti questi messaggi per loggetto mock, o per sopprimere il messaggio per metodi specifici aggiungendo EXPECT_CALL(...).Times(AnyNumber()). NON sopprimerlo aggiungendo ciecamente un EXPECT_CALL(...), altrimenti il test sarà difficile da manutenere.

10.3.2 Ignorare le Chiamate [Uninteresting]

Se non si è interessati a come viene chiamato un metodo mock, basta non dire nulla al riguardo. In questo caso, se il metodo viene chiamato, gMock eseguirà la sua azione di default per consentire al programma di test di continuare. Se non si è soddisfatti dellazione di default intrapresa da gMock, la si può sovrascrivere utilizzando DefaultValue<T>::Set() (descritto *qui*) o ON_CALL().

Si tenga presente che una volta espresso interesse per un particolare metodo mock (tramite EXPECT_CALL()), tutte le invocazioni ad esso devono corrispondere a qualche expectation. Se questa funzione viene chiamata ma gli argomenti non corrispondono ad alcuna istruzione EXPECT_CALL(), sarà un errore.

10.3.3 Evitare Chiamate [Unexpected]

Se un metodo mock non deve essere affatto chiamato, lo si dica esplicitamente:

```
using ::testing::_;
...
  EXPECT_CALL(foo, Bar(_))
      .Times(0);
```

Se alcune chiamate al metodo sono consentite, ma le altre no, elencare semplicemente tutte le chiamate previste:

```
using ::testing::AnyNumber;
using ::testing::Gt;
...
  EXPECT_CALL(foo, Bar(5));
  EXPECT_CALL(foo, Bar(Gt(10)))
      .Times(AnyNumber());
```

Una chiamata a foo.Bar() che non corrisponde a nessuna delle istruzioni EXPECT_CALL() sarà un errore.

10.3.4 Le Chiamate Poco Interessanti e Quelle Unexpected {#uninteresting-vs-unexpected}

Le chiamate *Uninteresting* e quelle *unexpected* sono concetti diversi in gMock. *Molto* diversi.

Una chiamata x.Y(...) è **uninteresting** se non è impostata *nemmeno una singola* EXPECT_CALL(x, Y(...)). In altre parole, il test non è affatto interessato al metodo x.Y(), come è evidente dal fatto che al test non interessa dire nulla al riguardo.

Una chiamata x.Y(...) è **unexpected** se è impostata *qualche* EXPECT_CALL(x, Y(...)), ma nessuna di esse corrisponde [match] alla chiamata. In altre parole, il test è interessato al metodo x.Y() (quindi imposta esplicitamente alcune EXPECT_CALL per verificare come viene chiamato); tuttavia, la verifica fallisce poiché il test non prevede [expect] che si verifichi questa particolare chiamata.

Una chiamata unexpected è sempre un errore, poiché il codice sottoposto a test non si comporta come il test si aspetta che si comporti.

Per default, una chiamata uninteresting non è un errore, poiché non viola alcun vincolo specificato dal test. (La filosofia di gMock è che non dire nulla significa che non ci sono vincoli). Tuttavia, porta a un warning, poiché *potrebbe* indicare un problema (ad esempio lautore del test potrebbe aver dimenticato di specificare un vincolo).

In gMock, `NiceMock` e `StrictMock` possono essere utilizzati per rendere una classe mock ńacondiscendenteż [nice] o ńrigorosaż [strict]. In che modo ciò influisce sulle chiamate uninteresting e le unexpected?

Un **nice mock** sopprime i *warnings* sulle chiamate uninteresting. È meno loquace del mock di default, ma per il resto è lo stesso. Se un test fallisce con un mock di default, fallirà anche utilizzando un mock ńniceż. E viceversa. Non ci si aspetti che rendere un mock ńniceż possa cambiare il risultato del test.

Un **strict mock** trasforma i warning sulle chiamate uninteresting in errori. Quindi rendere un mock strict può cambiare il risultato del test.

Diamo unocchiata ad un esempio:

```
TEST(...) {
  NiceMock<MockDomainRegistry> mock_registry;
  EXPECT_CALL(mock_registry, GetDomainOwner("google.com"))
          .WillRepeatedly(Return("Larry Page"));

  // Use mock_registry in code under test.
  ... &mock_registry ...
}
```

Lunico `EXPECT_CALL` qui dice che tutte le chiamate a `GetDomainOwner()` devono avere `"google.com"` come argomento. Se viene chiamata `GetDomainOwner("yahoo.com")`, si tratterà di una chiamata inaspettata [unexpected] e quindi di un errore. *Avere un mock nice non cambia la gravità di una chiamata unexpected.*

Allora come diciamo a gMock che `GetDomainOwner()` può essere chiamato anche con altri argomenti? La tecnica standard consiste nellaggiungere un ńacchiappa-tuttoż `EXPECT_CALL`:

```
  EXPECT_CALL(mock_registry, GetDomainOwner(_))
        .Times(AnyNumber());  // catches all other calls to this method.
  EXPECT_CALL(mock_registry, GetDomainOwner("google.com"))
        .WillRepeatedly(Return("Larry Page"));
```

Da tener presente che `_` è il carattere jolly che corrisponde a ńqualsiasi cosaż. Con questo, se viene chiamato `GetDomainOwner("google.com")`, farà ciò che dice il secondo `EXPECT_CALL`; se viene chiamato con un argomento diverso, farà quello che dice il primo `EXPECT_CALL`.

Notare che lordine dei due `EXPECT_CALL` è importante, poiché un `EXPECT_CALL` più recente ha la precedenza su quello precedente.

Per ulteriori informazioni su chiamate uninteresting, mock nice e mock strict, leggere *ńNice, Strict e Naggyż*.

10.3.5 Ignorare gli Argomenti Non Interessanti {#ParameterlessExpectations}

Se il test non si preoccupa dei parametri (si preoccupa solo del numero o dellordine delle chiamate), spesso si può semplicemente omettere lelenco dei parametri:

```
  // Expect foo.Bar( ... ) twice with any arguments.
  EXPECT_CALL(foo, Bar).Times(2);

  // Delegate to the given method whenever the factory is invoked.
  ON_CALL(foo_factory, MakeFoo)
      .WillByDefault(&BuildFooForTest);
```

Questa funzionalità è disponibile solo quando un metodo non è overloaded; per evitare comportamenti imprevisti, tentare di impostare unaspettativa [expectation] su un metodo in cui loverload specifico è ambiguo costituisce un errore di compilazione. Il problema si può aggirare fornendo una *interfaccia più semplice del mock* rispetto a quella fornita dalla classe mock-ata.

Questo pattern è utile anche quando gli argomenti sono interessanti, ma la logica della corrispondenza [match] è sostanzialmente complessa. Si può lasciare lelenco degli argomenti non specificato e utilizzare le azioni SaveArg per *salvare i valori per una successiva verifica*. Facendolo, si può facilmente distinguere la chiamata al metodo per il numero sbagliato di volte dalla chiamata con gli argomenti sbagliati.

10.3.6 Expect di Chiamate Ordinate {#OrderedCalls}

Sebbene unistruzione `EXPECT_CALL()` definita successivamente abbia la precedenza quando gMock tenta di far corrispondere [to match] una chiamata di funzione con una expectation, per default le chiamate non devono avvenire nellordine con cui sono scritte le istruzioni `EXPECT_CALL()`. Ad esempio, se gli argomenti corrispondono ai matcher nella seconda `EXPECT_CALL()`, ma non a quelli nella prima o nella terza, allora verrà utilizzata la seconda expectation.

Se si preferisce che tutte le chiamate avvengano nellordine previsto, inserire le istruzioni `EXPECT_CALL()` in un blocco in cui si definisce una variabile di tipo `InSequence`:

```
using ::testing::_;
using ::testing::InSequence;

  {
    InSequence s;

    EXPECT_CALL(foo, DoThis(5));
    EXPECT_CALL(bar, DoThat(_))
        .Times(2);
    EXPECT_CALL(foo, DoThis(6));
  }
```

In questo esempio, ci aspettiamo una chiamata a `foo.DoThis(5)`, seguita da due chiamate a `bar.DoThat()` dove largomento può essere qualsiasi cosa, che sono a turno seguite da una chiamata a `foo.DoThis(6)`. Se una chiamata è avvenuta fuori ordine, gMock segnalerà un errore.

10.3.7 Expecting Chiamate Parzialmente Ordinate {#PartialOrder}

A volte richiedere che tutto avvenga in un ordine predeterminato può portare a test fragili. Ad esempio, potremmo preoccuparci che `A` si presenti prima sia di `B` e `C`, ma non siamo interessati allordine relativo di `B` e `C`. In questo caso, il test dovrebbe riflettere il nostro reale intento, invece di essere eccessivamente vincolante.

gMock consente di imporre un DAG (directed acyclic graph) arbitrario sulle chiamate. Un modo per esprimere il DAG è utilizzare la *clausola* `After` di `EXPECT_CALL`.

Un altro modo è tramite la clausola `InSequence()` (diversa dalla classe `InSequence`), presa in prestito da jMock 2. È meno flessibile di `After()`, ma più conveniente quando si hanno lunghe catene di chiamate sequenziali, poiché non richiede di trovare nomi diversi per le expectation nelle catene. Ecco come funziona:

Se consideriamo le istruzioni `EXPECT_CALL()` come nodi in un grafo e aggiungiamo un arco dal nodo A al nodo B ovunque A debba trovarsi prima di B, possiamo ottenere un DAG. Usiamo il termine ńsequenzaż per indicare un percorso diretto in questo DAG. Ora, se scomponiamo il DAG in sequenze, dobbiamo solo sapere a quali sequenze appartiene ciascuna `EXPECT_CALL()` per poter ricostruire il DAG originale.

Quindi, per specificare lordine parziale sulle expectation dobbiamo fare due cose: prima definire degli oggetti `Sequence`, e poi per ciascuna `EXPECT_CALL()` dire si quale oggetto `Sequence` fa parte.

Le expectation nella stessa sequenza devono verificarsi nellordine in cui sono scritte. Per esempio,

```
using ::testing::Sequence;
...
  Sequence s1, s2;

  EXPECT_CALL(foo, A())
      .InSequence(s1, s2);
  EXPECT_CALL(bar, B())
      .InSequence(s1);
  EXPECT_CALL(bar, C())
      .InSequence(s2);
  EXPECT_CALL(foo, D())
      .InSequence(s2);
```

specifica il seguente DAG (dove `s1` è `A -> B` e `s2` è `A -> C -> D`):

```
      +---> B
      |
  A ---|
      |
      +---> C ---> D
```

Ciò significa che A deve verificarsi prima di B e C, e C deve verificarsi prima di D. Non ci sono restrizioni sullordine diverso da questi.

10.3.8 Controllare Quando una Expectation viene Ritirata

Quando viene chiamato un metodo mock, gMock considera solo le expectation ancora attive. Una expectation è attiva quando viene creata e diventa inattiva (ovvero *va in pensione*) quando si verifica una chiamata che deve avvenire in seguito. Ad esempio, in

```
using ::testing::_;
using ::testing::Sequence;
...
  Sequence s1, s2;

  EXPECT_CALL(log, Log(WARNING, _, "File too large."))      // #1
      .Times(AnyNumber())
      .InSequence(s1, s2);
  EXPECT_CALL(log, Log(WARNING, _, "Data set is empty."))   // #2
      .InSequence(s1);
  EXPECT_CALL(log, Log(WARNING, _, "User not found."))      // #3
      .InSequence(s2);
```

non appena il #2 o il #3 verranno abbinati [match], il #1 verrà ritirato. Se successivamente viene loggato un warning `"File too large."`, si tratterà di un errore.

Notare che una expectation viene ritirata automaticamente quando è saturata. Per esempio,

```
using ::testing::_;
...
  EXPECT_CALL(log, Log(WARNING, _, _));                     // #1
  EXPECT_CALL(log, Log(WARNING, _, "File too large."));     // #2
```

dice che ci sarà esattamente un warning con il messaggio `"File too large."`. Se anche il secondo warning contiene questo messaggio, #2 corrisponderà [match] nuovamente e genererà un errore di violazione del limite superiore [upper-bound-violated].

Se questo non è quello che si desidera, si può a chiedere una expectation ritirarsi non appena diventa satura:

```
using ::testing::_;
...
  EXPECT_CALL(log, Log(WARNING, _, _));                     // #1
  EXPECT_CALL(log, Log(WARNING, _, "File too large."))      // #2
      .RetiresOnSaturation();
```

Qui #2 può essere utilizzato solo una volta, quindi se si hanno due warning con il messaggio `"File too large."`, il primo corrisponderà [match] a #2 e il secondo corrisponderà a #1: non ci sarà errore.

10.4 Usare le Action

10.4.1 Restituire Riferimenti dai Metodi Mock

Se il tipo di ritorno di una funzione mock è un riferimento, si deve usare `ReturnRef()` anziché `Return()` per restituire un risultato:

```
using ::testing::ReturnRef;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(Bar&, GetBar, (), (override));
};
...
  MockFoo foo;
  Bar bar;
  EXPECT_CALL(foo, GetBar())
      .WillOnce(ReturnRef(bar));
...
```

10.4.2 Restituire Valori Live dai Metodi Mock

Lazione `Return(x)` salva una copia di `x` quando lazione viene creata e restituisce sempre lo stesso valore ogni volta che viene eseguita. A volte si vorrebbe restituire invece il valore *live* di `x` (ovvero il suo valore nel momento in cui lazione viene *eseguita*.). Si usa o `ReturnRef()` oppure `ReturnPointee()` per questo.

Se il tipo restituito dalla funzione mock è un riferimento, lo si può fare utilizzando `ReturnRef(x)`, come mostrato nella ricetta precedente (ńRestituire Riferimenti dai Metodi Mockż). Tuttavia, gMock non consente di utilizzare `ReturnRef()` in una funzione mock il cui tipo restituito non è un riferimento, poiché ciò di solito indica un errore dellutente. Allora, cosa si farà?

Anche se si potrebbe essere tentati, NON USARE `std::ref()`:

```
using ::testing::Return;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(int, GetValue, (), (override));
};
...
  int x = 0;
  MockFoo foo;
  EXPECT_CALL(foo, GetValue())
      .WillRepeatedly(Return(std::ref(x)));  // Wrong!
  x = 42;
  EXPECT_EQ(foo.GetValue(), 42);
```

Sfortunatamente, qui non funziona. Il codice precedente fallirà con errore:

```
Value of: foo.GetValue()
  Actual: 0
Expected: 42
```

Il motivo è che `Return(*value*)` converte `value` nel tipo restituito effettivo della funzione mock nel momento in cui lazione viene *creata*, non quando viene *eseguita*. (Questo comportamento è stato scelto perché lazione fosse sicura quando `value` è un oggetto proxy che fa riferimento ad alcuni oggetti temporanei). Di conseguenza, `std::ref(x)` è convertito in un valore `int` (anziché un `const int&`) quando la expectation è impostata e `Return(std::ref(x))` restituirà sempre 0.

`ReturnPointee(pointer)` è stato fornito per risolvere specificamente questo problema. Restituisce il valore puntato da `pointer` nel momento in cui lazione viene *eseguita*:

```
using ::testing::ReturnPointee;
...
  int x = 0;
  MockFoo foo;
  EXPECT_CALL(foo, GetValue())
      .WillRepeatedly(ReturnPointee(&x));  // Note the & here.
  x = 42;
  EXPECT_EQ(foo.GetValue(), 42);  // This will succeed now.
```

10.4.3 Combinazione di Azioni

Si vuol fare più di una cosa quando viene chiamata una funzione? Va bene. `DoAll()` consente di eseguire una sequenza di azioni ogni volta. Verrà utilizzato solo il valore restituito dellultima azione nella sequenza.

```
using ::testing::_;
using ::testing::DoAll;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(bool, Bar, (int n), (override));
};
...
  EXPECT_CALL(foo, Bar(_))
      .WillOnce(DoAll(action_1,
                      action_2,
                      ...
                      action_n));
```

Il valore restituito dellultima azione **deve** corrispondere al tipo restituito del metodo mock-ato. Nellesempio sopra, `action_n` potrebbe essere `Return(true)` o una lambda che restituisce un `bool`, ma non `SaveArg`, che restituisce `void`. Altrimenti la firma [signature] di `DoAll` non corrisponderebbe a quella prevista da `WillOnce`, che è la firma del metodo mock-ato, e non verrebbe compilato.

10.4.4 Verifica di argomenti Complessi {#SaveArgVerify}

Per verificare che un metodo venga chiamato con un argomento particolare ma i criteri di corrispondenza [match] sono complessi, può essere difficile distinguere tra errori di cardinalità (chiamare il metodo il numero sbagliato di volte) e gli errori di corrispondenza [match] dellargomento. Allo stesso modo, se si cè il ńmatchż di più parametri, potrebbe non essere facile distinguere quale argomento non è riuscito a corrispondere. Per esempio:

```
  // Not ideal: this could fail because of a problem with arg1 or arg2, or maybe
  // just the method wasn't called.
  EXPECT_CALL(foo, SendValues(_, ElementsAre(1, 4, 4, 7), EqualsProto( ... )));
```

Si possono invece salvare gli argomenti e testarli individualmente:

```
  EXPECT_CALL(foo, SendValues)
      .WillOnce(DoAll(SaveArg<1>(&actual_array), SaveArg<2>(&actual_proto)));
  ... run the test
  EXPECT_THAT(actual_array, ElementsAre(1, 4, 4, 7));
  EXPECT_THAT(actual_proto, EqualsProto( ... ));
```

10.4.5 Effetti Collaterali dei Mock {#MockingSideEffects}

A volte un metodo mostra il suo effetto non restituendo un valore ma tramite degli effetti collaterali. Ad esempio, potrebbe modificare alcuni stati globali o modificare un argomento di output. Per mock-are gli oggetti collaterali, in generale si può definire la propria azione implementando `::testing::ActionInterface`.

Se tutto ciò che si deve fare è modificare un argomento di output, conviene la ńactionż nativa `SetArgPointee()`:

```
using ::testing::_;
using ::testing::SetArgPointee;

class MockMutator : public Mutator {
 public:
  MOCK_METHOD(void, Mutate, (bool mutate, int* value), (override));
  ...
}
...
  MockMutator mutator;
  EXPECT_CALL(mutator, Mutate(true, _))
      .WillOnce(SetArgPointee<1>(5));
```

In questo esempio, quando viene chiamato `mutator.Mutate()`, assegneremo 5 alla variabile `int` puntata dallargomento #1 (in base 0).

`SetArgPointee()` crea comodamente una copia interna del valore che gli si passa, eliminando la necessità di mantenere il valore nello scope e in vita [alive]. Limplicazione tuttavia è che il valore deve avere un costruttore di copia e un operatore di assegnazione.

Se anche il metodo mock deve restituire un valore, si può concatenare `SetArgPointee()` con `Return()` usando `DoAll()`, ricordando di mettere listruzione `Return()` in ultimo:

```
using ::testing::_;
using ::testing::DoAll;
using ::testing::Return;
using ::testing::SetArgPointee;

class MockMutator : public Mutator {
 public:
  ...
  MOCK_METHOD(bool, MutateInt, (int* value), (override));
}
...
  MockMutator mutator;
  EXPECT_CALL(mutator, MutateInt(_))
      .WillOnce(DoAll(SetArgPointee<0>(5),
                      Return(true)));
```

Notare, tuttavia, che se si usa il metodo `ReturnOKWith()`, esso sovrascriverà [override] i valori forniti da `SetArgPointee()` nei parametri di risposta della chiamata di funzione.

Se largomento di output è un array, si usa invece lazione `SetArrayArgument<N>(first, last)`. Questa copia gli elementi nellintervallo di origine `[first, last)` nellarray puntato dallargomento `N`-esimo (in base 0):

```
using ::testing::NotNull;
using ::testing::SetArrayArgument;

class MockArrayMutator : public ArrayMutator {
 public:
  MOCK_METHOD(void, Mutate, (int* values, int num_values), (override));
  ...
```

(continues on next page)

(continua dalla pagina precedente)

```
}
...
  MockArrayMutator mutator;
  int values[5] = {1, 2, 3, 4, 5};
  EXPECT_CALL(mutator, Mutate(NotNull(), 5))
      .WillOnce(SetArrayArgument<0>(values, values + 5));
```

Funziona anche quando largomento è un iteratore di output:

```
using ::testing::_;
using ::testing::SetArrayArgument;

class MockRolodex : public Rolodex {
 public:
  MOCK_METHOD(void, GetNames, (std::back_insert_iterator<vector<string>>),
              (override));
  ...
}
...
  MockRolodex rolodex;
  vector<string> names = {"George", "John", "Thomas"};
  EXPECT_CALL(rolodex, GetNames(_))
      .WillOnce(SetArrayArgument<0>(names.begin(), names.end()));
```

10.4.6 Modifica del Comportamento di un Oggetto Mock in Base allo Stato

Se si prevede che una chiamata modifichi il comportamento di un oggetto mock, si può usare `::testing::InSequence` per specificare comportamenti diversi prima e dopo la chiamata:

```
using ::testing::InSequence;
using ::testing::Return;

...
  {
     InSequence seq;
     EXPECT_CALL(my_mock, IsDirty())
         .WillRepeatedly(Return(true));
     EXPECT_CALL(my_mock, Flush());
     EXPECT_CALL(my_mock, IsDirty())
         .WillRepeatedly(Return(false));
  }
  my_mock.FlushIfDirty();
```

Questo fa sì che `my_mock.IsDirty()` restituisca `true` prima che `my_mock.Flush()` venga chiamato e restituisca `false` successivamente.

Se la modifica del comportamento è più complessa, se ne possono memorizzare gli effetti in una variabile e fare in modo che un metodo mock ottenga il valore restituito da quella variabile:

```
using ::testing::_;
using ::testing::SaveArg;
using ::testing::Return;

ACTION_P(ReturnPointee, p) { return *p; }
...
  int previous_value = 0;
  EXPECT_CALL(my_mock, GetPrevValue)
```

(continues on next page)

(continua dalla pagina precedente)

```
      .WillRepeatedly(ReturnPointee(&previous_value));
  EXPECT_CALL(my_mock, UpdateValue)
      .WillRepeatedly(SaveArg<0>(&previous_value));
  my_mock.DoSomethingToUpdateValue();
```

Qui `my_mock.GetPrevValue()` restituirà sempre largomento dellultima chiamata `UpdateValue()`.

10.4.7 Impostazione del Valore di Default per un Tipo Restituito {#DefaultValue}

Se il tipo restituito di un metodo mock è un tipo o puntatore C++ nativo, per default restituirà 0 quando richiamato. Inoltre, in C++ 11 e versioni successive, un metodo mock il cui tipo restituito ha un costruttore di default, restituirà di default un valore costruito per default. Si deve solo specificare unazione se questo valore di default non corrisponde alle proprie esigenze.

A volte, si vuole modificare questo valore di default o specificare un valore di default per i tipi di cui gMock non è a conoscenza. Lo si può fare utilizzando la classe template `::testing::DefaultValue`:

```
using ::testing::DefaultValue;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(Bar, CalculateBar, (), (override));
};


...
  Bar default_bar;
  // Sets the default return value for type Bar.
  DefaultValue<Bar>::Set(default_bar);

  MockFoo foo;

  // We don't need to specify an action here, as the default
  // return value works for us.
  EXPECT_CALL(foo, CalculateBar());

  foo.CalculateBar();  // This should return default_bar.

  // Unsets the default return value.
  DefaultValue<Bar>::Clear();
```

Notare che la modifica del valore di default per un tipo può rendere difficile la comprensione dei test. Consigliamo di utilizzare questa funzione con giudizio. Ad esempio, per assicurarsi che le chiamate `Set()` e `Clear()` siano proprio accanto al codice che utilizza il mock.

10.4.8 Impostazione delle Azioni di Default per un Metodo Mock

Abbiamo imparato come modificare il valore di default di un determinato tipo. Tuttavia, questo potrebbe essere troppo grossolano per i propri scopi: forse ci sono due metodi mock con lo stesso tipo di ritorno e si vuole che abbiano comportamenti diversi. La macro `ON_CALL()` consente di personalizzare il comportamento del mock a livello del metodo:

```
using ::testing::_;
using ::testing::AnyNumber;
using ::testing::Gt;
using ::testing::Return;
...
```

(continues on next page)

(continua dalla pagina precedente)

```
  ON_CALL(foo, Sign(_))
      .WillByDefault(Return(-1));
  ON_CALL(foo, Sign(0))
      .WillByDefault(Return(0));
  ON_CALL(foo, Sign(Gt(0)))
      .WillByDefault(Return(1));

  EXPECT_CALL(foo, Sign(_))
      .Times(AnyNumber());

  foo.Sign(5);   // This should return 1.
  foo.Sign(-9);  // This should return -1.
  foo.Sign(0);   // This should return 0.
```

Come intuito, quando ci sono più istruzioni `ON_CALL()`, quelle più recenti nellordine hanno la precedenza su quelle più vecchie. In altre parole, verrà utilizzato l**ultimo** che corrisponde agli argomenti della funzione. Questo ordine di corrispondenza consente di impostare il comportamento comune nel costruttore di un oggetto mock o nella fase di impostazione della fixture e di specializzare il comportamento del mock in seguito.

Notare che sia `ON_CALL` che `EXPECT_CALL` hanno la stessa regola ńle istruzioni successive hanno la precedenzaż, ma non interagiscono. Cioè, le `EXPECT_CALL` hanno il proprio ordine di precedenza distinto da quello di `ON_CALL`.

10.4.9 Utilizzo di Funzioni/Metodi/Funtori/Lambda come Azioni {#FunctionsAsActions}

Se le azioni native non soddisfano, si può utilizzare un chiamabile [callable] esistente (funzione, `std::function`, metodo, funtore, lambda) come azione.

```
using ::testing::_; using ::testing::Invoke;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(int, Sum, (int x, int y), (override));
  MOCK_METHOD(bool, ComplexJob, (int x), (override));
};

int CalculateSum(int x, int y) { return x + y; }
int Sum3(int x, int y, int z) { return x + y + z; }

class Helper {
 public:
  bool ComplexJob(int x);
};

...
  MockFoo foo;
  Helper helper;
  EXPECT_CALL(foo, Sum(_, _))
      .WillOnce(&CalculateSum)
      .WillRepeatedly(Invoke(NewPermanentCallback(Sum3, 1)));
  EXPECT_CALL(foo, ComplexJob(_))
      .WillOnce(Invoke(&helper, &Helper::ComplexJob))
      .WillOnce([] { return true; })
      .WillRepeatedly([](int x) { return x > 0; });

  foo.Sum(5, 6);         // Invokes CalculateSum(5, 6).
  foo.Sum(2, 3);         // Invokes Sum3(1, 2, 3).
```

(continues on next page)

(continua dalla pagina precedente)

```
  foo.ComplexJob(10);    // Invokes helper.ComplexJob(10).
  foo.ComplexJob(-1);    // Invokes the inline lambda.
```

Lunico requisito è che il tipo della funzione, ecc. deve essere *compatibile* con la firma della funzione mock, il che significa che gli argomenti di questultima (se ne accetta qualcuno) possono essere convertiti implicitamente negli argomenti corrispondenti del primo e il tipo restituito del primo possono essere convertiti implicitamente in quello del secondo. Quindi, si può invocare qualcosa il cui tipo *non* sia esattamente lo stesso della funzione mock, purché sia sicuro farlo: bello, eh?

Notare che:

- Lazione assume la proprietà [ownership] della callback e la eliminerà [delete] quando lazione stessa verrà distrutta.
- Se il tipo della callback deriva da un tipo di callback di base `C`, è necessario implicitamente eseguire il cast a `C` per risolvere loverload, ad es.

```
using ::testing::Invoke;
...
  ResultCallback<bool>* is_ok = ...;
  ... Invoke(is_ok) ...;  // This works.

  BlockingClosure* done = new BlockingClosure;
  ... Invoke(implicit_cast<Closure*>(done)) ...;  // The cast is necessary.
```

10.4.10 Uso di Funzioni con Informazioni Extra come Azioni

La funzione o il funtore che si chiama utilizzando `Invoke()` deve avere lo stesso numero di argomenti della funzione mock per cui la si usa. A volte si potrebbe avere una funzione che accetta più argomenti e si è disposti a passare manualmente gli argomenti extra per colmare il divario. In gMock lo si può fare utilizzando le callback con argomenti pre-associati [pre-bound]. Ecco un esempio:

```
using ::testing::Invoke;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(char, DoThis, (int n), (override));
};

char SignOfSum(int x, int y) {
  const int sum = x + y;
  return (sum > 0) ? '+' : (sum < 0) ? '-' : '0';
}

TEST_F(FooTest, Test) {
  MockFoo foo;

  EXPECT_CALL(foo, DoThis(2))
      .WillOnce(Invoke(NewPermanentCallback(SignOfSum, 5)));
  EXPECT_EQ(foo.DoThis(2), '+');  // Invokes SignOfSum(5, 2).
}
```

10.4.11 Invcare una Funzione/Metodo/Funtore/Lambda/Callback Senza Argomenti

`Invoke()` passa gli argomenti della funzione mock alla funzione, ecc. invocata in modo tale che il chiamato abbia lintero contesto della chiamata con cui lavorare.. Se la funzione invocata non è interessata ad alcuni o a tutti gli argomenti, può semplicemente ignorarli.

Tuttavia, un pattern comune è che lautore di un test desidera invocare una funzione senza gli argomenti della funzione mock. Potrebbe farlo utilizzando una funzione wrapper che elimina gli argomenti prima di invocare una [nullary function]. Inutile dire che questo può essere noioso e oscurare lo scopo del test.

Ci sono due soluzioni a questo problema. Innanzitutto, si può passare qualsiasi ńcallableż di zero arg come azione. In alternativa, si usa `InvokeWithoutArgs()`, che è come `Invoke()` tranne per il fatto che non passa gli argomenti della funzione fittizia al chiamato. Ecco un esempio di ciascuno:

```
using ::testing::_;
using ::testing::InvokeWithoutArgs;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(bool, ComplexJob, (int n), (override));
};

bool Job1() { ... }
bool Job2(int n, char c) { ... }

...
  MockFoo foo;
  EXPECT_CALL(foo, ComplexJob(_))
      .WillOnce([] { Job1(); });
      .WillOnce(InvokeWithoutArgs(NewPermanentCallback(Job2, 5, 'a')));

  foo.ComplexJob(10);  // Invokes Job1().
  foo.ComplexJob(20);  // Invokes Job2(5, 'a').
```

Notare che:

- Lazione assume la proprietà [ownership] della callback e la eliminerà [delete] quando lazione stessa verrà distrutta.
- Se il tipo della callback deriva da un tipo di callback di base `C`, è necessario implicitamente eseguire il cast a `C` per risolvere loverload, ad es.

  ```
  using ::testing::InvokeWithoutArgs;
  ...
    ResultCallback<bool>* is_ok = ...;
    ... InvokeWithoutArgs(is_ok) ...;  // This works.

    BlockingClosure* done = ...;
    ... InvokeWithoutArgs(implicit_cast<Closure*>(done)) ...;
    // The cast is necessary.
  ```

10.4.12 Invocare un Argomento della Funzione Mock

A volte una funzione mock riceverà un puntatore a funzione, un funtore (in altre parole, un ńcallableż) come argomento, ad es.

```
class MockFoo : public Foo {
 public:
  MOCK_METHOD(bool, DoThis, (int n, (ResultCallback1<bool, int>* callback)),
              (override));
};
```

e si potrebbe invocare questo argomento richiamabile:

```
using ::testing::_;
...
  MockFoo foo;
  EXPECT_CALL(foo, DoThis(_, _))
      .WillOnce(...);
      // Will execute callback->Run(5), where callback is the
      // second argument DoThis() receives.
```

{: .callout .note} NOTA: La sezione seguente è la documentazione di prima che il C++ avesse le lambda:

Arghh, si deve fare riferimento a un argomento di funzione mock ma il C++ non ha (ancora) le lambda, quindi si deve definire la propria azione. :-(Oh! veramente?

Beh, gMock ha unazione per risolvere *esattamente* questo problema:

```
InvokeArgument<N>(arg_1, arg_2, ..., arg_m)
```

invocherà lN-esimo (0-based) argomento ricevuto dalla funzione mock, con `arg_1`, `arg_2`, , e `arg_m`. Non importa se largomento è un puntatore a funzione, un funtore o una callback. gMock li gestisce tutti.

Con ciò, si potrebbe scrivere:

```
using ::testing::_;
using ::testing::InvokeArgument;
...
  EXPECT_CALL(foo, DoThis(_, _))
      .WillOnce(InvokeArgument<1>(5));
      // Will execute callback->Run(5), where callback is the
      // second argument DoThis() receives.
```

Cosa succede se il ńcallableż accetta un argomento per riferimento? Nessun problema: basta racchiuderlo allinterno di `std::ref()`:

```
  ...
  MOCK_METHOD(bool, Bar,
              ((ResultCallback2<bool, int, const Helper&>* callback)),
              (override));
  ...
  using ::testing::_;
  using ::testing::InvokeArgument;
  ...
  MockFoo foo;
  Helper helper;
  ...
  EXPECT_CALL(foo, Bar(_))
      .WillOnce(InvokeArgument<0>(5, std::ref(helper)));
      // std::ref(helper) guarantees that a reference to helper, not a copy of
      // it, will be passed to the callback.
```

Cosa succede se il callable accetta un argomento per riferimento e **non** racchiudiamo largomento in `std::ref()`? In questo caso `InvokeArgument()` *creerà una copia* dellargomento e passerà un *riferimento alla copia*, invece di un riferimento al valore originale, al richiamabile [callable]. Ciò è particolarmente utile quando largomento è un valore temporaneo:

```
  ...
  MOCK_METHOD(bool, DoThat, (bool (*f)(const double& x, const string& s)),
              (override));
  ...
  using ::testing::_;
```

(continues on next page)

(continua dalla pagina precedente)

```
  using ::testing::InvokeArgument;
  ...
  MockFoo foo;
  ...
  EXPECT_CALL(foo, DoThat(_))
      .WillOnce(InvokeArgument<0>(5.0, string("Hi")));
      // Will execute (*f)(5.0, string("Hi")), where f is the function pointer
      // DoThat() receives.  Note that the values 5.0 and string("Hi") are
      // temporary and dead once the EXPECT_CALL() statement finishes.  Yet
      // it's fine to perform this action later, since a copy of the values
      // are kept inside the InvokeArgument action.
```

10.4.13 Ignorare un Risultato di unAzione

A volte si ha unazione che restituisce *qualcosa*, ma cè bisogno di unazione che restituisca `void` (forse per usarla in una funzione mock che restituisca `void`, o forse deve essere utilizzato in `DoAll()` e non è lultimo della lista). `IgnoreResult()` consente di farlo. Per esempio:

```
using ::testing::_;
using ::testing::DoAll;
using ::testing::IgnoreResult;
using ::testing::Return;

int Process(const MyData& data);
string DoSomething();

class MockFoo : public Foo {
 public:
  MOCK_METHOD(void, Abc, (const MyData& data), (override));
  MOCK_METHOD(bool, Xyz, (), (override));
};

  ...
  MockFoo foo;
  EXPECT_CALL(foo, Abc(_))
      // .WillOnce(Invoke(Process));
      // The above line won't compile as Process() returns int but Abc() needs
      // to return void.
      .WillOnce(IgnoreResult(Process));
  EXPECT_CALL(foo, Xyz())
      .WillOnce(DoAll(IgnoreResult(DoSomething),
                      // Ignores the string DoSomething() returns.
                      Return(true)));
```

Notare che **non si può** usare `IgnoreResult()` su unazione che restituisce già `void`. Ciò porterà a brutti errori del compilatore.

10.4.14 Selezionare gli Argomenti di unAzione {#SelectingArgs}

Supponiamo che si abbia una funzione mock `Foo()` che accetta sette argomenti e che si abbia unazione personalizzata che da invocare quando viene chiamata `Foo()`. Il problema è che lazione personalizzata richiede solo tre argomenti:

```
using ::testing::_;
using ::testing::Invoke;
...
```

(continues on next page)

(continua dalla pagina precedente)

```
  MOCK_METHOD(bool, Foo,
              (bool visible, const string& name, int x, int y,
               (const map<pair<int, int>>), double& weight, double min_weight,
               double max_wight));
...
bool IsVisibleInQuadrant1(bool visible, int x, int y) {
  return visible && x >= 0 && y >= 0;
}
...
  EXPECT_CALL(mock, Foo)
      .WillOnce(Invoke(IsVisibleInQuadrant1));  // Uh, won't compile. :-(
```

Per compiacere il dio del compilatore, si deve definire un ńadaptorż che abbia la stessa firma di `Foo()` e si chiama lazione personalizzata con gli argomenti giusti:

```
using ::testing::_;
using ::testing::Invoke;
...
bool MyIsVisibleInQuadrant1(bool visible, const string& name, int x, int y,
                            const map<pair<int, int>, double>& weight,
                            double min_weight, double max_wight) {
  return IsVisibleInQuadrant1(visible, x, y);
}
...
  EXPECT_CALL(mock, Foo)
      .WillOnce(Invoke(MyIsVisibleInQuadrant1));  // Now it works.
```

Ma non è imbarazzante?

gMock fornisce un *action adaptor* generico, così ci si può dedicare agli affari più importanti piuttosto che scrivere i propri adaptor. Ecco la sintassi:

```
WithArgs<N1, N2, ..., Nk>(action)
```

crea unazione che passa gli argomenti della funzione mock agli indici dati (in base 0) all`action` interna e la esegue. Utilizzando `WithArgs`, lesempio originale può essere scritto come:

```
using ::testing::_;
using ::testing::Invoke;
using ::testing::WithArgs;
...
  EXPECT_CALL(mock, Foo)
      .WillOnce(WithArgs<0, 2, 3>(Invoke(IsVisibleInQuadrant1)));  // No need to␣
↪define your own adaptor.
```

Per una migliore leggibilità, gMock offre anche:

- `WithoutArgs(action)` quando l`action` interna accetta *nessun* argomento, e
- `WithArg<N>(action)` (nessuna `s` dopo `Arg`) quando l`action` interna prende *un* argomento.

Si comprenderà che `InvokeWithoutArgs(...)` è solo ńzucchero sintatticoż per `WithoutArgs(Invoke(...))`.

Ecco ulteriori suggerimenti:

- Lazione interna utilizzata in `WithArgs` e simili non deve essere necessariamente `Invoke()`: può essere qualsiasi cosa.
- Se necessario è possibile ripetere un argomento nellelenco, ad es. `WithArgs<2, 3, 3, 5>(...)`.
- Si può cambiare lordine degli argomenti, p.es. `WithArgs<3, 2, 1>(...)`.

- I tipi degli argomenti selezionati *non* devono corrispondere esattamente alla firma dellazione interna. Funziona finché possono essere convertiti implicitamente negli argomenti corrispondenti dellazione interna. Ad esempio, se il quarto argomento della funzione mock è un `int` e `my_action` accetta un `double`, `WithArg<4>(my_action)` funzionerà.

10.4.15 Ignorare gli Argomenti nella Funzioni Azioni

La ricetta *selecting-an-actions-arguments* ci ha mostrato un modo per far combaciare una funzione mock e unazione con elenchi di argomenti incompatibili. Lo svantaggio è che racchiudere lazione in `WithArgs<...>()` può diventare noioso per le persone che scrivono i test.

Se si sta definendo una funzione (o metodo, funtore, lambda, callback) da utilizzare con `Invoke*()` e non si è interessati ad alcuni dei suoi argomenti, unalternativa a `WithArgs` consiste nel dichiarare gli argomenti non interessanti come `Unused`. Ciò rende la definizione meno confusa e meno fragile nel caso in cui cambino i tipi di argomenti poco interessanti. Potrebbe anche aumentare la possibilità che la funzione dellazione possa essere riutilizzata. Per esempio, dato

```
 public:
  MOCK_METHOD(double, Foo, double(const string& label, double x, double y),
              (override));
  MOCK_METHOD(double, Bar, (int index, double x, double y), (override));
```

invece di

```
using ::testing::_;
using ::testing::Invoke;

double DistanceToOriginWithLabel(const string& label, double x, double y) {
  return sqrt(x*x + y*y);
}
double DistanceToOriginWithIndex(int index, double x, double y) {
  return sqrt(x*x + y*y);
}
...
  EXPECT_CALL(mock, Foo("abc", _, _))
      .WillOnce(Invoke(DistanceToOriginWithLabel));
  EXPECT_CALL(mock, Bar(5, _, _))
      .WillOnce(Invoke(DistanceToOriginWithIndex));
```

si può scrivere

```
using ::testing::_;
using ::testing::Invoke;
using ::testing::Unused;

double DistanceToOrigin(Unused, double x, double y) {
  return sqrt(x*x + y*y);
}
...
  EXPECT_CALL(mock, Foo("abc", _, _))
      .WillOnce(Invoke(DistanceToOrigin));
  EXPECT_CALL(mock, Bar(5, _, _))
      .WillOnce(Invoke(DistanceToOrigin));
```

10.4.16 Condivisione di Azioni

Proprio come i matcher, un oggetto azione gMock è costituito da un puntatore a un oggetto di implementazione con conteggio dei riferimenti. Pertanto anche le azioni di copia sono consentite e molto efficienti. Quando lultima azione che fa riferimento alloggetto di implementazione muore, loggetto di implementazione verrà eliminato.

Se si ha unazione complessa che da utilizzare più e più volte, si potrebbe non doverla creare da zero ogni volta. Se lazione non ha uno stato interno (cioè se fa sempre la stessa cosa indipendentemente da quante volte è stata chiamata), si può assegnarla a una variabile di azione e utilizzare quella variabile ripetutamente. Per esempio:

```
using ::testing::Action;
using ::testing::DoAll;
using ::testing::Return;
using ::testing::SetArgPointee;
...
  Action<bool(int*)> set_flag = DoAll(SetArgPointee<0>(5),
                                      Return(true));
  ... use set_flag in .WillOnce() and .WillRepeatedly() ...
```

Tuttavia, se lazione ha un proprio stato, si potrebbe rimanere sorpresi se si condivide loggetto dellazione. Supponiamo di avere una action factory `IncrementCounter(init)` che crea unazione che incrementa e restituisce un contatore il cui valore iniziale è `init`, utilizzando due azioni create dalla stessa espressione e utilizzando unazione condivisa mostrerà comportamenti diversi. Esempio:

```
  EXPECT_CALL(foo, DoThis())
      .WillRepeatedly(IncrementCounter(0));
  EXPECT_CALL(foo, DoThat())
      .WillRepeatedly(IncrementCounter(0));
  foo.DoThis();  // Returns 1.
  foo.DoThis();  // Returns 2.
  foo.DoThat();  // Returns 1 - DoThat() uses a different
                 // counter than DoThis()'s.
```

contro

```
using ::testing::Action;
...
  Action<int()> increment = IncrementCounter(0);
  EXPECT_CALL(foo, DoThis())
      .WillRepeatedly(increment);
  EXPECT_CALL(foo, DoThat())
      .WillRepeatedly(increment);
  foo.DoThis();  // Returns 1.
  foo.DoThis();  // Returns 2.
  foo.DoThat();  // Returns 3 - the counter is shared.
```

10.4.17 Testare il Comportamento Asincrono

Un problema spesso riscontrato con gMock è che può essere difficile testare il comportamento asincrono. Supponiamo di avere una classe `EventQueue` da testare e che sia stata creata uninterfaccia `EventDispatcher` separata in modo da poterla facilmente renderla mock. Tuttavia, limplementazione della classe ha attivato tutti gli eventi su un thread in background, rendendo difficile la tempistica dei test. Si potrebbero semplicemente inserire le istruzioni `sleep()` e sperare per il meglio, ma ciò rende il comportamento del test non deterministico. Un modo migliore è quello di utilizzare le azioni gMock e gli oggetti `Notification` per forzare il test asincrono a comportarsi in modo sincrono.

```
class MockEventDispatcher : public EventDispatcher {
  MOCK_METHOD(bool, DispatchEvent, (int32), (override));
};
```

(continues on next page)

(continua dalla pagina precedente)

```
TEST(EventQueueTest, EnqueueEventTest) {
  MockEventDispatcher mock_event_dispatcher;
  EventQueue event_queue(&mock_event_dispatcher);

  const int32 kEventId = 321;
  absl::Notification done;
  EXPECT_CALL(mock_event_dispatcher, DispatchEvent(kEventId))
      .WillOnce([&done] { done.Notify(); });

  event_queue.EnqueueEvent(kEventId);
  done.WaitForNotification();
}
```

Nellesempio sopra, impostiamo le nostre normali expectation gMock, ma poi aggiungiamo una-zione aggiuntiva per notificare loggetto `Notification`. Ora possiamo semplicemente chiamare `Notification::WaitForNotification()` nel thread principale per attendere il completamento della chiamata asincrona. Successivamente, la nostra suite di test è completa e possiamo uscire in sicurezza.

{: .callout .note} Nota: questo esempio ha uno svantaggio: ovvero, se la expectation non viene soddisfatta, il test verrà eseguito allinfinito. Alla fine andrà in timeout e fallirà, ma richiederà più tempo e sarà leggermente più difficile eseguire il debug. Per alleviare questo problema, si può utilizzare `WaitForNotificationWithTimeout(ms)` invece di `WaitForNotification()`.

10.5 Ricette Varie sullUso di gMock

10.5.1 Mock di Metodi Che Usano Tipi Move-Only

Il C++11 ha introdotto i *tipi move-only* (tipi di solo spostamento). Un valore ńmove-only-typedż può essere spostato da un oggetto a un altro, ma non può essere copiato. `std::unique_ptr<T>` è probabilmente il tipo di solo spostamento [move-only] più comunemente utilizzato.

Il mock di un metodo che accetta e/o restituisce tipi move-only presenta alcune sfide, ma nulla di insormontabile. Questa ricetta mostra come lo si può fare. Notare che il supporto per gli argomenti move-only di metodi è stato introdotto in gMock solo nellaprile 2017; nel codice più vecchio si trovano dei *workarounds* più complessi per la mancanza di questa funzionalità.

Diciamo che stiamo lavorando a un progetto immaginario che consente di pubblicare e condividere frammenti chiamati buzzes. Il codice utilizza questi tipi:

```
enum class AccessLevel { kInternal, kPublic };

class Buzz {
 public:
  explicit Buzz(AccessLevel access) { ... }
  ...
};

class Buzzer {
 public:
  virtual ~Buzzer() {}
  virtual std::unique_ptr<Buzz> MakeBuzz(StringPiece text) = 0;
  virtual bool ShareBuzz(std::unique_ptr<Buzz> buzz, int64_t timestamp) = 0;
  ...
};
```

Un oggetto Buzz rappresenta un frammento [snippet] pubblicato. Una classe che implementa linterfaccia Buzzer è in grado di creare e condividere dei Buzz. I metodi in Buzzer possono restituire un unique_ptr<Buzz> o ricevere un unique_ptr<Buzz>. Ora dobbiamo mock-are Buzzer nei nostri test.

Per il mock di un metodo che accetta o restituisce tipi move-only, si usa semplicemente la familiare sintassi MOCK_METHOD:

```
class MockBuzzer : public Buzzer {
 public:
  MOCK_METHOD(std::unique_ptr<Buzz>, MakeBuzz, (StringPiece text), (override));
  MOCK_METHOD(bool, ShareBuzz, (std::unique_ptr<Buzz> buzz, int64_t timestamp),
              (override));
};
```

Ora che abbiamo definito la classe mock, possiamo usarla nei test. Nei seguenti esempi di codice, assumiamo di aver definito un oggetto MockBuzzer chiamato mock_buzzer_:

```
  MockBuzzer mock_buzzer_;
```

Per prima cosa vediamo come possiamo impostare le expectation sul metodo MakeBuzz(), che restituisce un unique_ptr<Buzz>.

Come al solito, se si imposta una expectation senza unazione (ad esempio la clausola .WillOnce() o la .WillRepeatedly()), quando si attiva la expectation, lazione di default per quel metodo verrà intrapresa. Poiché unique_ptr<> ha un costruttore di default che restituisce un unique_ptr nullo, questo è ciò che si ottiene se non si specifica unazione:

```
using ::testing::IsNull;
...
  // Use the default action.
  EXPECT_CALL(mock_buzzer_, MakeBuzz("hello"));

  // Triggers the previous EXPECT_CALL.
  EXPECT_THAT(mock_buzzer_.MakeBuzz("hello"), IsNull());
```

Se non si è soddisfatti dellazione di default, la si può modificare come al solito; vedere *Impostazione delle Azioni di Default*.

Se cè solo da restituire un valore move-only, lo si può usare in combinazione con WillOnce. Per esempio:

```
  EXPECT_CALL(mock_buzzer_, MakeBuzz("hello"))
      .WillOnce(Return(std::make_unique<Buzz>(AccessLevel::kInternal)));
  EXPECT_NE(nullptr, mock_buzzer_.MakeBuzz("hello"));
```

Tempo di quiz! Cosa accadrà se unazione Return viene eseguita più di una volta (ad esempio scrivendo ... .WillRepeatedly(Return(std::move(...)));)? Rifletteteci, dopo la prima volta che viene eseguita lazione, il valore sorgente verrà consumato (poiché è un valore di solo spostamento [move-only]), quindi la volta successiva, non cè alcun valore da spostare - si otterrà un errore a run-time secondo cui Return(std::move(...)) può essere eseguito solo una volta.

Se cè bisogno che il metodo mock faccia qualcosa di più del semplice spostamento di un valore di default, ricordarsi che si può sempre utilizzare un oggetto lambda o un richiamabile [callable], che può fare praticamente tutto:

```
  EXPECT_CALL(mock_buzzer_, MakeBuzz("x"))
      .WillRepeatedly([](StringPiece text) {
        return std::make_unique<Buzz>(AccessLevel::kInternal);
      });

  EXPECT_NE(nullptr, mock_buzzer_.MakeBuzz("x"));
  EXPECT_NE(nullptr, mock_buzzer_.MakeBuzz("x"));
```

Ogni volta che questa EXPECT_CALL si attiva, verrà creato e restituito un nuovo unique_ptr<Buzz>. Non lo si può fare con Return(std::make_unique<...>(...)).

Ciò riguarda la restituzione di valori move-only; ma come lavoriamo con metodi che accettano argomenti move-only? La risposta è che funzionano normalmente, anche se alcune azioni non verranno compilate quando uno qualsiasi degli argomenti del metodo è move-only. Si può sempre usare Return, o una *lambda o un funtore*:

```
using ::testing::Unused;

EXPECT_CALL(mock_buzzer_, ShareBuzz(NotNull(), _)).WillOnce(Return(true));
EXPECT_TRUE(mock_buzzer_.ShareBuzz(std::make_unique<Buzz>(AccessLevel::kInternal)),
            0);

EXPECT_CALL(mock_buzzer_, ShareBuzz(_, _)).WillOnce(
    [](std::unique_ptr<Buzz> buzz, Unused) { return buzz != nullptr; });
EXPECT_FALSE(mock_buzzer_.ShareBuzz(nullptr, 0));
```

Molte azioni native (WithArgs, WithoutArgs,DeleteArg, SaveArg,) potrebbero in linea di principio supportare argomenti move-only, ma il supporto per questo non è ancora implementato. Se questo è bloccante, segnalare un bug.

Alcune azioni (ad esempio DoAll) copiano i loro argomenti internamente, quindi non possono mai funzionare con oggetti non copiabili; si dovranno usare invece i funtori.

Workarounds legacy per i tipi move-only {#LegacyMoveOnly}

Il supporto per gli argomenti move-only di funzioni è stato introdotto in gMock solo nellaprile del 2017. Nel codice precedente, si potrebbe riscontrare la seguente soluzione alternativa per la mancanza di questa funzionalità (non è più necessaria: la includiamo solo come riferimento):

```
class MockBuzzer : public Buzzer {
 public:
  MOCK_METHOD(bool, DoShareBuzz, (Buzz* buzz, Time timestamp));
  bool ShareBuzz(std::unique_ptr<Buzz> buzz, Time timestamp) override {
    return DoShareBuzz(buzz.get(), timestamp);
  }
};
```

Il trucco sta nel delegare il metodo ShareBuzz() a un metodo mock (chiamiamolo DoShareBuzz()) che non accetta parametri move-only. Poi, invece di impostare le expectation su ShareBuzz(), si impostano su l metodo mock DoShareBuzz():

```
  MockBuzzer mock_buzzer_;
  EXPECT_CALL(mock_buzzer_, DoShareBuzz(NotNull(), _));

  // When one calls ShareBuzz() on the MockBuzzer like this, the call is
  // forwarded to DoShareBuzz(), which is mocked.  Therefore this statement
  // will trigger the above EXPECT_CALL.
  mock_buzzer_.ShareBuzz(std::make_unique<Buzz>(AccessLevel::kInternal), 0);
```

10.5.2 Velocizzare la Compilazione

Che ci si creda o meno, la *stragrande maggioranza* del tempo impiegato nella compilazione di una classe mock è nella generazione del suo costruttore e del distruttore, poiché eseguono compiti non banali (ad esempio la verifica delle expectation). Inoltre, i metodi mock con firme diverse hanno tipi diversi e quindi i loro costruttori/distruttori devono essere generati separatamente dal compilatore. Di conseguenza, se ci sono molti tipi diversi di metodi mock, la compilazione della classe mock può diventare molto lenta.

Se si riscontra una compilazione lenta, si può spostare la definizione del costruttore e del distruttore della classe mock fuori dal corpo della classe e in un file .cc. In questo modo, anche se si #include la classe mock in N file,

il compilatore dovrà generare il suo costruttore e distruttore solo una volta, ne risulterà una compilazione molto più veloce.

Illustriamo lidea con un esempio. Ecco la definizione di una classe mock prima di applicare questa ricetta:

```
// File mock_foo.h.
...
class MockFoo : public Foo {
 public:
  // Since we don't declare the constructor or the destructor,
  // the compiler will generate them in every translation unit
  // where this mock class is used.

  MOCK_METHOD(int, DoThis, (), (override));
  MOCK_METHOD(bool, DoThat, (const char* str), (override));
  ... more mock methods ...
};
```

Dopo la modifica, sarebbe simile a:

```
// File mock_foo.h.
...
class MockFoo : public Foo {
 public:
  // The constructor and destructor are declared, but not defined, here.
  MockFoo();
  virtual ~MockFoo();

  MOCK_METHOD(int, DoThis, (), (override));
  MOCK_METHOD(bool, DoThat, (const char* str), (override));
  ... more mock methods ...
};
```

e

```
// File mock_foo.cc.
#include "path/to/mock_foo.h"

// The definitions may appear trivial, but the functions actually do a
// lot of things through the constructors/destructors of the member
// variables used to implement the mock methods.
MockFoo::MockFoo() {}
MockFoo::~MockFoo() {}
```

10.5.3 Forzare una Verifica

Quando verrà distrutto, loggetto mock verificherà automaticamente che tutte le sue expectation siano state soddisfatte e, in caso contrario, genererà errori di googletest. Questo è conveniente perché lascia con una cosa in meno di cui preoccuparsi. Cioè, a meno che non si sia sicuri che loggetto mock verrà distrutto.

Come è possibile che loggetto mock alla fine non venga distrutto? Ebbene, potrebbe essere creato nellheap e posseduto [own] dal codice che si sta testando Supponiamo che ci sia un bug in quel codice e che non si effettui correttamente il delete delloggetto mock: si potrebbe finire con un test positivo quando in realtà cè un bug.

Lutilizzo di un ńheap checkerż è una buona idea e può alleviare i problemi, ma la sua implementazione non è affidabile al 100%. Quindi, a volte si vuole *forzare* gMock a verificare un oggetto mock prima che venga (si spera) distrutto. Lo si può fare con `Mock::VerifyAndClearExpectations(&mock_object)`:

```
TEST(MyServerTest, ProcessesRequest) {
  using ::testing::Mock;

  MockFoo* const foo = new MockFoo;
  EXPECT_CALL(*foo, ...)...;
  // ... other expectations ...

  // server now owns foo.
  MyServer server(foo);
  server.ProcessRequest(...);

  // In case that server's destructor will forget to delete foo,
  // this will verify the expectations anyway.
  Mock::VerifyAndClearExpectations(foo);
}  // server is destroyed when it goes out of scope here.
```

{: .callout .tip} **Tip:** La funzione `Mock::VerifyAndClearExpectations()` restituisce un `bool` per indicare se la verifica ha avuto esito positivo (`true` per sì), quindi si può racchiudere la chiamata alla funzione allinterno di `ASSERT_TRUE()` se non ha senso andare oltre in caso quando la verifica non è riuscita.

Non stabilire nuove expectation dopo aver verificato e ripulito un mock dopo il suo utilizzo. Limpostazione delle expectation dopo il codice che esercita il mock ha un comportamento indefinito. Vedere Uso dei Mock nei Test per ulteriori informazioni.

10.5.4 Uso dei Checkpoint {#UsingCheckPoints}

A volte si deve testare il comportamento di un oggetto mock in fasi le cui dimensioni sono gestibili, oppure si devono impostare expectation più dettagliate su quali chiamate API richiamano quali funzioni mock.

Una tecnica utilizzabile è quella di mettere le expectation in una sequenza e inserire chiamate a una funzione fittizia di ńcheckpointż in punti specifici. Poi si può verificare che le chiamate alle funzioni mock avvengano al momento giusto. Ad esempio, per testare il codice:

```
  Foo(1);
  Foo(2);
  Foo(3);
```

e per verificare che `Foo(1)` e `Foo(3)` invochino entrambe `mock.Bar("a")`, ma che `Foo(2)` non invoca nulla, si può scrivere:

```
using ::testing::MockFunction;

TEST(FooTest, InvokesBarCorrectly) {
  MyMock mock;
  // Class MockFunction<F> has exactly one mock method.  It is named
  // Call() and has type F.
  MockFunction<void(string check_point_name)> check;
  {
    InSequence s;

    EXPECT_CALL(mock, Bar("a"));
    EXPECT_CALL(check, Call("1"));
    EXPECT_CALL(check, Call("2"));
    EXPECT_CALL(mock, Bar("a"));
  }
  Foo(1);
  check.Call("1");
  Foo(2);
```

(continues on next page)

(continua dalla pagina precedente)

```
  check.Call("2");
  Foo(3);
}
```

La specifica della expectation dice che la prima chiamata `Bar("a")` deve avvenire prima del checkpoint ńlż, la seconda chiamata `Bar("a")` deve avvenire dopo il checkpoint ńż, e tra i due checkpoint non dovrebbe succedere nulla. I checkpoint espliciti chiariscono quale `Bar("a")` viene chiamata da quale chiamata a `Foo()`.

10.5.5 Mock dei Distruttori

A volte si vuol verificare che un oggetto mock venga distrutto al momento giusto, ad es. dopo la chiamata a `bar->A()` ma prima di chiamare `bar->B()`. Sappiamo già che è possibile specificare vincoli sull*ordine* delle chiamate alla funzione mock, quindi tutto ciò che dobbiamo fare è avere il mock del distruttore della funzione mock.

Sembra semplice, tranne che per un problema: un distruttore è una funzione speciale con una sintassi e una semantica speciali e la macro `MOCK_METHOD` non funziona per questo:

```
MOCK_METHOD(void, ~MockFoo, ());  // Won't compile!
```

La buona notizia è che si può usare un semplice pattern per ottenere lo stesso effetto. Innanzitutto, si aggiunge una funzione mock `Die()` alla classe mock e la si chiama nel distruttore, in questo modo:

```
class MockFoo : public Foo {
  ...
  // Add the following two lines to the mock class.
  MOCK_METHOD(void, Die, ());
  ~MockFoo() override { Die(); }
};
```

(Se il nome `Die()` entra in conflitto con un simbolo esistente, si sceglie un altro nome). Ora, abbiamo tradotto il problema di testare quando un oggetto `MockFoo` ńmuoreż nel testare quando viene chiamato il metodo `Die()`:

```
  MockFoo* foo = new MockFoo;
  MockBar* bar = new MockBar;
  ...
  {
    InSequence s;

    // Expects *foo to die after bar->A() and before bar->B().
    EXPECT_CALL(*bar, A());
    EXPECT_CALL(*foo, Die());
    EXPECT_CALL(*bar, B());
  }
```

E questo è tutto.

10.5.6 Usare gMock e i Thread {#UsingThreads}

In una **unit** test, è meglio isolare e testare una parte di codice in un contesto a thread singolo. Ciò evita condizioni di ńraceż e ńdead lockż e semplifica molto il debug del test.

Eppure la maggior parte dei programmi sono multi-thread e talvolta per testare qualcosa abbiamo bisogno di lavorarci sopra da più di un thread. gMock funziona anche per questo scopo.

Si ricordano i passaggi per utilizzare un mock:

1. Si crea un oggetto mock `foo`.
2. Se ne impostano le azioni e le expectation con `ON_CALL()` e con `EXPECT_CALL()`.

3. Il codice da testare chiama i metodi di `foo`.
4. Facoltativamente, verificare e reimpostare il mock.
5. Distruggere il mock manualmente o lasciare che sia il codice sotto test a farlo. Il distruttore lo verificherà automaticamente.

Seguendo queste semplici regole, i mock e i thread possono convivere felicemente:

- Si esegue il *codice di test* (in contrapposizione al codice da testare) in *un solo* thread. Ciò facilita il test da eseguire.
- Ovviamente si può fare il passo #1 senza il locking.
- Quando si eseguono i passi #2 e #5, nessun altro thread deve accedere a `foo`. Anchesso ovvio, huh?
- #3 e #4 possono essere eseguiti in un thread o in più thread, - come si vuole. gMock si occupa del locking, quindi non si deve fare nulla, a meno che non sia richiesto dalla logica del test.

Se si violano le regole (ad esempio, se si impostano le expectation su un mock mentre un altro thread ne chiama i metodi), si otterrà un comportamento indefinito. Questo non è bello, quindi non farlo.

gMock garantisce che lazione per una funzione mock venga eseguita nello stesso thread che ha chiamato la funzione mock. Ad esempio, in

```
EXPECT_CALL(mock, Foo(1))
    .WillOnce(action1);
EXPECT_CALL(mock, Foo(2))
    .WillOnce(action2);
```

se `Foo(1)` viene chiamato nel thread 1 e `Foo(2)` viene chiamato nel thread 2, gMock eseguirà `action1` nel thread 1 e `action2` nel thread 2.

gMock *non* impone una sequenza sulle azioni eseguite in thread diversi (farlo potrebbe creare situazioni di stallo poiché le azioni potrebbero dover cooperare). Ciò significa che lesecuzione di `action1` e di `action2` nellesempio precedente *può* interlacciarsi. Se questo è un problema, si deve aggiungere una logica di sincronizzazione adeguata ad `action1` e ad `action2` per rendere il test thread-safe.

Ricordarsi, inoltre, che `DefaultValue<T>` è una risorsa globale che potenzialmente influenza *tutti* gli oggetti mock attivi nel programma. Naturalmente, non ci si vorrà ingarbugliare con più thread o quando ci sono ancora mock in azione.

10.5.7 Controllare la Quantità di Informazioni che Scrive gMock

Quando gMock vede qualcosa che potrebbe essere un errore (ad esempio viene chiamata una funzione mock senza una expectation, ovvero una chiamata poco interessante, che è consentita ma forse si è dimenticato di vietare esplicitamente la chiamata), stampa alcuni messaggi di warning, incluso gli argomenti della funzione, il valore restituito e il trace dello stack. Si spera che questo ricordi di dare unocchiata e vedere se cè davvero un problema.

A volte si è sicuri che i test siano corretti e si potrebbe non apprezzare messaggi così amichevoli. Altre volte, si sta eseguendo il debug dei test o si sta imparando il comportamento del codice in esame e si vorrebbe poter osservare ogni chiamata mock che avviene (inclusi i valori degli argomenti, il valore restituito e il trace dello stack). Chiaramente, una sola taglia non va bene per tutti.

Si può controllare quanto gMock stampa utilizzando il flag della riga di comando `--gmock_verbose=LEVEL`, dove `LEVEL` è una stringa con tre possibili valori:

- `info`: gMock stamperà tutti i messaggi informativi, i warning e gli errori (i più dettagliati). Con questa impostazione, gMock loggerà anche tutte le chiamate alle macro `ON_CALL/EXPECT_CALL`. Includerà uno stack trace nei warning ńuninteresting callż.
- `warning`: gMock stamperà sia i warning che gli errori (meno dettagliato); ometterà gli stack trace nei warning ńuninteresting callż. Questo è il default.
- `error`: gMock stamperà solo gli errori (i meno dettagliati).

In alternativa, si può aggiustare il valore del flag dai test in questo modo:

```
  ::testing::FLAGS_gmock_verbose = "error";
```

Se si trova che gMock stampa troppi stack frame con i suoi messaggi informativi o di warning, se ne può controllare la quantità con il flag `--gtest_stack_trace_depth=max_depth` flag.

Ora, usare giudiziosamente il flag giusto per consentire a gMock di servire al meglio!

10.5.8 Ottenere la Supervisione nelle Chiamate Mock

Abbiamo un test che usa gMock. Fallisce: gMock ti dice che alcune expectation non sono soddisfatte. Tuttavia, non si è sicuri del perché: cè un errore di battitura da qualche parte nei matcher? È sbagliato lordine delle `EXPECT_CALL`? Oppure il codice sotto test sta facendo qualcosa di sbagliato? Come scoprirne la causa?

Non sarebbe bello se si avesse la vista a raggi X e si potessi effettivamente vedere la traccia di tutte le `EXPECT_CALL` e delle chiamate dei metodo mock mentre vengono effettuate? Per ogni chiamata, si potrebbero vedere i valori effettivi degli argomenti e a quale `EXPECT_CALL` gMock crede che corrisponda [match]? Se cè ancora bisogno di aiuto per capire chi ha effettuato queste chiamate, potrebbe essere utile poter vedere lo stack trace completo ad ogni chiamata mock?

Si può sbloccare questo potere eseguendo il test con il flag `--gmock_verbose=info`. Ad esempio, dato il programma di test:

```
#include <gmock/gmock.h>

using ::testing::_;
using ::testing::HasSubstr;
using ::testing::Return;

class MockFoo {
 public:
  MOCK_METHOD(void, F, (const string& x, const string& y));
};

TEST(Foo, Bar) {
  MockFoo mock;
  EXPECT_CALL(mock, F(_, _)).WillRepeatedly(Return());
  EXPECT_CALL(mock, F("a", "b"));
  EXPECT_CALL(mock, F("c", HasSubstr("d")));

  mock.F("a", "good");
  mock.F("a", "b");
}
```

se lo si esegue con `--gmock_verbose=info`, si vedrà questo output:

```
[ RUN      ] Foo.Bar

foo_test.cc:14: EXPECT_CALL(mock, F(_, _)) invoked
Stack trace: ...

foo_test.cc:15: EXPECT_CALL(mock, F("a", "b")) invoked
Stack trace: ...

foo_test.cc:16: EXPECT_CALL(mock, F("c", HasSubstr("d"))) invoked
Stack trace: ...

foo_test.cc:14: Mock function call matches EXPECT_CALL(mock, F(_, _))...
    Function call: F(@0x7fff7c8dad40"a",@0x7fff7c8dad10"good")
Stack trace: ...
```

(continues on next page)

(continua dalla pagina precedente)

```
foo_test.cc:15: Mock function call matches EXPECT_CALL(mock, F("a", "b"))...
    Function call: F(@0x7fff7c8dada0"a",@0x7fff7c8dad70"b")
Stack trace: ...

foo_test.cc:16: Failure
Actual function call count doesn't match EXPECT_CALL(mock, F("c", HasSubstr("d")))...
         Expected: to be called once
           Actual: never called - unsatisfied and active
[  FAILED  ] Foo.Bar
```

Supponiamo che il bug sia che la `"c"` nella terza `EXPECT_CALL` sia un errore di battitura e in realtà dovrebbe essere `"a"`. Col messaggio precedente, si vedrebbe che leffettiva chiamata `F("a", "good")` corrisponde alla prima `EXPECT_CALL`, non alla terza come si pensava. Da ciò dovrebbe essere ovvio che il terzo `EXPECT_CALL` è scritto in modo errato. Caso risolto.

Se si è interessati alla call trace dei mock ma non a quelle dello stack, si può combinare `--gmock_verbose=info` con `--gtest_stack_trace_depth=0` sulla riga di comando del test.

10.5.9 Esecuzione di Test in Emacs

Se si creano ed eseguono i test in Emacs utilizzando il comando `M-x google-compile` (come fanno molti utenti di googletest), le posizioni dei sorgenti di gMock e gli errori di googletest verranno evidenziati. Basta premere `<Enter>` su uno di essi e si verrà indirizzati alla riga incriminata. Oppure si può semplicemente digitare `C-x` per passare allerrore successivo.

Per semplificarlo ulteriormente, si possono aggiungere le seguenti righe al file `~/.emacs`:

```
(global-set-key "\M-m"  'google-compile)  ; m is for make
(global-set-key [M-down] 'next-error)
(global-set-key [M-up]  '(lambda () (interactive) (next-error -1)))
```

Poi si può digitare `M-m` per avviare una build (se si vuole eseguire anche il test, assicurarsi solo che `foo_test.run` o `runtests` sia nel comando di compilazione fornito dopo aver digitato `M-m`, o `M-up`/`M-down` per spostarsi avanti e indietro tra gli errori.

10.6 Estendere gMock

10.6.1 Scrivere Rapidamente Nuovi Matcher {#NewMatchers}

{: .callout .warning} ATTENZIONE: gMock non garantisce quando o quante volte un matcher verrà invocato. Pertanto, tutti i matcher devono essere funzionalmente puri. Consultare *questa sezione* per ulteriori dettagli.

La famiglia delle macro `MATCHER*` può essere utilizzata per definire facilmente dei matcher custom. La sintassi:

```
MATCHER(name, description_string_expression) { statements; }
```

definirà un matcher con il nome dato che esegue le istruzioni, che deve restituire un `bool` per indicare se la corrispondenza ha successo. Allinterno delle istruzioni, si può fare riferimento al valore a cui corrisponde `arg` e fare riferimento al suo tipo tramite `arg_type`.

La *description string* è unespressione di tipo `string` che documenta ciò che fa il matcher e viene utilizzata per generare il messaggio di errore quando la corrispondenza fallisce. Può (e dovrebbe) fare riferimento alla variabile `bool` speciale `negation` e dovrebbe valutare la descrizione del matcher quando `negation` è `false`, o quello della negazione del matcher quando `negation` è `true`.

Per comodità, supponiamo che la stringa della descrizione sia vuota (`""`), nel qual caso gMock utilizzerà la sequenza di parole nel nome del matcher come descrizione.

Basic Example

```
MATCHER(IsDivisibleBy7, "") { return (arg % 7) == 0; }
```

consente di scrivere

```
  // Expects mock_foo.Bar(n) to be called where n is divisible by 7.
  EXPECT_CALL(mock_foo, Bar(IsDivisibleBy7()));
```

o,

```
  using ::testing::Not;
  ...
  // Verifies that a value is divisible by 7 and the other is not.
  EXPECT_THAT(some_expression, IsDivisibleBy7());
  EXPECT_THAT(some_other_expression, Not(IsDivisibleBy7()));
```

Se le affermazioni di cui sopra falliscono, stamperanno qualcosa come:

```
  Value of: some_expression
  Expected: is divisible by 7
    Actual: 27
  ...
  Value of: some_other_expression
  Expected: not (is divisible by 7)
    Actual: 21
```

dove le descrizioni `"is divisible by 7"` e `"not (is divisible by 7)"` vengono calcolate automaticamente dal nome del matcher `IsDivisibleBy7`.

Adding Custom Failure Messages

Come notato, le descrizioni generate automaticamente (specialmente quelle per la negazione) potrebbero non essere così eccezionali. Si possono sempre sovrascriverle con unespressione `string` a scelta:

```
MATCHER(IsDivisibleBy7,
        absl::StrCat(negation ? "isn't" : "is", " divisible by 7")) {
  return (arg % 7) == 0;
}
```

Facoltativamente, si possono trasmettere informazioni aggiuntive a un argomento nascosto denominato `result_listener` per spiegare il risultato della corrispondenza [match]. Ad esempio, una definizione migliore di `IsDivisibleBy7` è:

```
MATCHER(IsDivisibleBy7, "") {
  if ((arg % 7) == 0)
    return true;

  *result_listener << "the remainder is " << (arg % 7);
  return false;
}
```

Con questa definizione, laffermazione di cui sopra darà un messaggio migliore:

```
  Value of: some_expression
  Expected: is divisible by 7
    Actual: 27 (the remainder is 6)
```

Using EXPECT_ Statements in Matchers

You can also use `EXPECT_...` statements inside custom matcher definitions. In many cases, this allows you to write your matcher more concisely while still providing an informative error message. Per esempio:

```
MATCHER(IsDivisibleBy7, "") {
  const auto remainder = arg % 7;
  EXPECT_EQ(remainder, 0);
  return true;
}
```

If you write a test that includes the line `EXPECT_THAT(27, IsDivisibleBy7());`, you will get an error something like the following:

```
Expected equality of these values:
  remainder
    Which is: 6
  0
```

MatchAndExplain

Si dovrebbe consentire a `MatchAndExplain()` di stampare *qualsiasi informazione aggiuntiva* che possa aiutare un utente a comprendere il risultato della corrispondenza [match]. Notare che dovrebbe spiegare perché la corrispondenza ha successo in caso di successo (a meno che non sia ovvio) - questo è utile quando il matcher viene utilizzato allinterno di `Not()`. Non è necessario stampare il valore dellargomento stesso, poiché gMock lo stampa già.

Argument Types

The type of the value being matched (`arg_type`) is determined by the context in which you use the matcher and is supplied to you by the compiler, so you dont need to worry about declaring it (nor can you). Ciò consente al matcher di essere polimorfico. Ad esempio, `IsDivisibleBy7()` può essere utilizzato per corrispondere [match] a qualsiasi tipo in cui il valore di `(arg % 7) == 0` può essere convertito implicitamente in un `bool`. Nellesempio `Bar(IsDivisibleBy7())` sopra, se il metodo `Bar()` accetta un `int`, `arg_type` sarà `int`; se richiede un `unsigned long`, `arg_type` sarà `unsigned long`; e così via.

10.6.2 Scrivere Rapidamente Nuovi Matcher Parametrizzati

A volte servirà un matcher che abbia parametri. Per questo si può usare la macro:

```
MATCHER_P(name, param_name, description_string) { statements; }
```

dove la description string può essere `""` o unespressione `string` che fa riferimento a `negation` e a `param_name`.

Per esempio:

```
MATCHER_P(HasAbsoluteValue, value, "") { return abs(arg) == value; }
```

consentirà di scrivere:

```
  EXPECT_THAT(Blah("a"), HasAbsoluteValue(n));
```

che può portare a questo messaggio (assumendo che `n` sia 10):

```
  Value of: Blah("a")
  Expected: has absolute value 10
    Actual: -9
```

Notare che vengono stampati sia la descrizione del matcher che il suo parametro, rendendo il messaggio di facile comprensione.

Nel corpo della definizione del matcher, si può scrivere `foo_type` per fare riferimento al tipo di un parametro chiamato `foo`. Ad esempio, nel corpo di `MATCHER_P(HasAbsoluteValue, value)` sopra, si può scrivere `value_type` per fare riferimento al tipo di `value`.

gMock fornisce anche `MATCHER_P2`, `MATCHER_P3`, , fino a `MATCHER_P10` per supportare i matcher multiparametro:

```
MATCHER_Pk(name, param_1, ..., param_k, description_string) { statements; }
```

Notare che la stringa di descrizione personalizzata è per una particolare *istanza* del matcher, in cui i parametri sono stati associati a valori effettivi. Pertanto di solito i valori dei parametri faranno parte della descrizione. gMock consente di farlo facendo riferimento ai parametri del matcher nellespressione della stringa della descrizione.

Per esempio,

```
using ::testing::PrintToString;
MATCHER_P2(InClosedRange, low, hi,
           absl::StrFormat("%s in range [%s, %s]", negation ? "isn't" : "is",
                           PrintToString(low), PrintToString(hi))) {
  return low <= arg && arg <= hi;
}
...
EXPECT_THAT(3, InClosedRange(4, 6));
```

genererebbe un errore che contiene il messaggio:

```
  Expected: is in range [4, 6]
```

Specificando "" come descrizione, il messaggio di errore conterrà la sequenza di parole nel nome del matcher seguita dai valori dei parametri stampati come una tupla. Per esempio,

```
  MATCHER_P2(InClosedRange, low, hi, "") { ... }
  ...
  EXPECT_THAT(3, InClosedRange(4, 6));
```

genererebbe un errore che contiene il testo:

```
  Expected: in closed range (4, 6)
```

Ai fini della tipizzazione, è possibile visualizzare

```
MATCHER_Pk(Foo, p1, ..., pk, description_string) { ... }
```

come abbreviazione di

```
template <typename p1_type, ..., typename pk_type>
FooMatcherPk<p1_type, ..., pk_type>
Foo(p1_type p1, ..., pk_type pk) { ... }
```

Scrivendo `Foo(v1, ..., vk)`, il compilatore deduce i tipi dei parametri `v1`, , e `vk` autonomamente. Se non soddisfa il risultato dellinferenza del tipo, si possono specificare i tipi istanziando esplicitamente il template, come in `Foo<long, bool>(5, false)`. Come detto in precedenza, non si può (né si deve) specificare `arg_type` poiché viene determinato dal contesto in cui viene utilizzato il matcher.

Si può assegnare il risultato dellespressione `Foo(p1, ..., pk)` ad una variabile di tipo `FooMatcherPk<p1_type, ..., pk_type>`. Questo può essere utile quando si compongono i matcher. I matcher che non hanno un parametro o ne hanno solo uno hanno tipi speciali: si può assegnare `Foo()` a una variabile di tipo `FooMatcher` e assegnare `Foo(p)` a una variabile di tipo `FooMatcherP<p_type>`.

Sebbene sia possibile creare unistanza di un template matcher con tipi riferimento, il passaggio dei parametri tramite puntatore in genere rende il codice più leggibile. Se però si deve comunque passare un parametro per riferimento, si tenga presente che nel messaggio di errore generato dal matcher si vedrà il valore delloggetto referenziato ma non il suo indirizzo.

Si può avere loverload dei matcher con diversi numeri di parametri:

```
MATCHER_P(Blah, a, description_string_1) { ... }
MATCHER_P2(Blah, a, b, description_string_2) { ... }
```

Sebbene sia forte la tentazione di utilizzare sempre le macro `MATCHER*` quando si definisce un nuovo matcher, di dovrebbe anche considerare di implementare direttamente linterfaccia del matcher (vedere le ricette che seguono), soprattutto se cè bisogno di usare molto il matcher. Sebbene questi approcci richiedano più lavoro, offrono un maggiore controllo sui tipi del valore da abbinare e sui parametri del matcher, il che in generale porta a messaggi di errore del compilatore migliori che ripagano nel lungo periodo. Consentono inoltre loverloading di matcher in base ai tipi di parametri (invece che basarsi solo sul numero di parametri).

10.6.3 Scrivere Nuovi Matcher Monomorfici

A matcher of type `testing::Matcher<T>` implements the matcher interface for `T` and does two things: it tests whether a value of type `T` matches the matcher, and can describe what kind of values it matches. Questultima capacità viene utilizzata per generare messaggi di errore leggibili quando le expectation vengono violate. Some matchers can even explain why it matches or doesnt match a certain value, which can be helpful when the reason isnt obvious.

Because a matcher of type `testing::Matcher<T>` for a particular type `T` can only be used to match a value of type `T`, we call it ńmonomorphic.ż

Un matcher di `T` deve dichiarare un typedef come:

```
using is_gtest_matcher = void;
```

e supporta le seguenti operazioni:

```
// Match a value and optionally explain into an ostream.
bool matched = matcher.MatchAndExplain(value, maybe_os);
// where `value` is of type `T` and
// `maybe_os` is of type `std::ostream*`, where it can be null if the caller
// is not interested in there textual explanation.

matcher.DescribeTo(os);
matcher.DescribeNegationTo(os);
// where `os` is of type `std::ostream*`.
```

Se cè bisogno di un matcher personalizzato ma `Truly()` non è una buona opzione (ad esempio, non soddisfa il modo in cui `Truly(predicate)` si descrive, oppure per rendere il matcher polimorfico come lo è `Eq(value)`), si può definire un matcher in due passaggi: prima si implementa linterfaccia del matcher e poi si definisce una funzione factory per creare unistanza del matcher. Il secondo passaggio non è strettamente necessario ma rende più gradevole la sintassi dellutilizzo del matcher.

Ad esempio, si può definire un matcher per verificare se un `int` è divisibile per 7 e poi usarlo in questo modo:

```
using ::testing::Matcher;

class DivisibleBy7Matcher {
 public:
  using is_gtest_matcher = void;

  bool MatchAndExplain(int n, std::ostream*) const {
    return (n % 7) == 0;
  }

  void DescribeTo(std::ostream* os) const {
    *os << "is divisible by 7";
  }
```

(continues on next page)

(continua dalla pagina precedente)

```
  void DescribeNegationTo(std::ostream* os) const {
    *os << "is not divisible by 7";
  }
};

Matcher<int> DivisibleBy7() {
  return DivisibleBy7Matcher();
}

...
  EXPECT_CALL(foo, Bar(DivisibleBy7()));
```

Il messaggio del matcher è migliorabile trasmettendo informazioni aggiuntive allargomento `os` in `MatchAndExplain()`:

```
class DivisibleBy7Matcher {
 public:
  bool MatchAndExplain(int n, std::ostream* os) const {
    const int remainder = n % 7;
    if (remainder != 0 && os != nullptr) {
      *os << "the remainder is " << remainder;
    }
    return remainder == 0;
  }
  ...
};
```

Poi, `EXPECT_THAT(x, DivisibleBy7());` può generare un messaggio come questo:

```
Value of: x
Expected: is divisible by 7
  Actual: 23 (the remainder is 2)
```

{: .callout .tip} Tip: per comodità, `MatchAndExplain()` può accettare un `MatchResultListener*` invece di `std::ostream*`.

10.6.4 Scrivere Nuovi Matcher Polimorfici

Unlike a monomorphic matcher, which can only be used to match a value of a particular type, a *polymorphic* matcher is one that can be used to match values of multiple types. For example, `Eq(5)` is a polymorhpic matcher as it can be used to match an `int`, a `double`, a `float`, and so on. You should think of a polymorphic matcher as a *matcher factory* as opposed to a `testing::Matcher<SomeType>` - itself is not an actual matcher, but can be implicitly converted to a `testing::Matcher<SomeType>` depending on the context.

Expanding what we learned above to polymorphic matchers is now as simple as adding templates in the right place.

```
class NotNullMatcher {
 public:
  using is_gtest_matcher = void;

  // To implement a polymorphic matcher, we just need to make MatchAndExplain a
  // template on its first argument.

  // In this example, we want to use NotNull() with any pointer, so
  // MatchAndExplain() accepts a pointer of any type as its first argument.
  // In general, you can define MatchAndExplain() as an ordinary method or
```

(continues on next page)

(continua dalla pagina precedente)

```
  // a method template, or even overload it.
  template <typename T>
  bool MatchAndExplain(T* p, std::ostream*) const {
    return p != nullptr;
  }

  // Describes the property of a value matching this matcher.
  void DescribeTo(std::ostream* os) const { *os << "is not NULL"; }

  // Describes the property of a value NOT matching this matcher.
  void DescribeNegationTo(std::ostream* os) const { *os << "is NULL"; }
};

NotNullMatcher NotNull() {
  return NotNullMatcher();
}

...

  EXPECT_CALL(foo, Bar(NotNull()));  // The argument must be a non-NULL pointer.
```

10.6.5 Implementazione di Matcher Legacy

La definizione dei matcher era un po più complicata, in quanto richiedeva diverse classi di supporto e funzioni virtuali. Per implementare un matcher per il tipo `T` utilizzando lAPI legacy si deve derivare da `MatcherInterface<T>` e chiamare `MakeMatcher` per costruire loggetto.

Linterfaccia è simile alla seguente:

```
class MatchResultListener {
 public:
  ...
  // Streams x to the underlying ostream; does nothing if the ostream
  // is NULL.
  template <typename T>
  MatchResultListener& operator<<(const T& x);

  // Returns the underlying ostream.
  std::ostream* stream();
};

template <typename T>
class MatcherInterface {
 public:
  virtual ~MatcherInterface();

  // Returns true if and only if the matcher matches x; also explains the match
  // result to 'listener'.
  virtual bool MatchAndExplain(T x, MatchResultListener* listener) const = 0;

  // Describes this matcher to an ostream.
  virtual void DescribeTo(std::ostream* os) const = 0;

  // Describes the negation of this matcher to an ostream.
  virtual void DescribeNegationTo(std::ostream* os) const;
};
```

Fortunatamente, la maggior parte delle volte si può definire facilmente un matcher polimorfico con laiuto di `MakePolymorphicMatcher()`. Ecco un esempio di come si possa definire `NotNull()`:

```
using ::testing::MakePolymorphicMatcher;
using ::testing::MatchResultListener;
using ::testing::PolymorphicMatcher;

class NotNullMatcher {
 public:
  // To implement a polymorphic matcher, first define a COPYABLE class
  // that has three members MatchAndExplain(), DescribeTo(), and
  // DescribeNegationTo(), like the following.

  // In this example, we want to use NotNull() with any pointer, so
  // MatchAndExplain() accepts a pointer of any type as its first argument.
  // In general, you can define MatchAndExplain() as an ordinary method or
  // a method template, or even overload it.
  template <typename T>
  bool MatchAndExplain(T* p,
                       MatchResultListener* /* listener */) const {
    return p != NULL;
  }

  // Describes the property of a value matching this matcher.
  void DescribeTo(std::ostream* os) const { *os << "is not NULL"; }

  // Describes the property of a value NOT matching this matcher.
  void DescribeNegationTo(std::ostream* os) const { *os << "is NULL"; }
};

// To construct a polymorphic matcher, pass an instance of the class
// to MakePolymorphicMatcher().  Note the return type.
PolymorphicMatcher<NotNullMatcher> NotNull() {
  return MakePolymorphicMatcher(NotNullMatcher());
}

...

  EXPECT_CALL(foo, Bar(NotNull()));  // The argument must be a non-NULL pointer.
```

{: .callout .note} **Nota:** La classe matcher polimorfica **non** ha bisogno di ereditare da `MatcherInterface` o qualsiasi altra classe, e i suoi metodi **non** devono essere virtuali.

Come in un matcher monomorfico, si può spiegare il risultato della corrispondenza trasmettendo informazioni aggiuntive allargomento `listener` in `MatchAndExplain()`.

10.6.6 Implementing Composite Matchers {#CompositeMatchers}

Sometimes we want to define a matcher that takes other matchers as parameters. For example, `DistanceFrom(target, m)` is a polymorphic matcher that takes a matcher `m` as a parameter. It tests that the distance from `target` to the value being matched satisfies sub-matcher `m`.

If you are implementing such a composite matcher, youll need to generate the description of the matcher based on the description(s) of its sub-matcher(s). You can see the implementation of `DistanceFrom()` in `googlemock/include/gmock/gmock-matchers.h` for an example. In particular, pay attention to `DistanceFromMatcherImpl`. Notice that it stores the sub-matcher as a `const Matcher<const Distance&> distance_matcher_` instead of a polymorphic matcher - this allows it to call `distance_matcher_.DescribeTo(os)` to describe the sub-matcher. If the sub-matcher is stored as a polymorphic matcher instead, it would not be possible to get its description as in general polymorphic matchers dont know how to describe them-

selves - they are matcher factories instead of actual matchers; only after being converted to `Matcher<SomeType>` can they be described.

10.6.7 Scrivere Nuove Cardinalità

Una cardinalità viene utilizzata in `Times()` per dire a gMock quante volte ci si aspetta che avvenga una chiamata. Non deve essere esatta. Ad esempio, si può dire `AtLeast(5)` o `Between(2, 4)`.

Se linsieme nativo delle cardinalità non è sufficiente, se ne può definire uno implementando la seguente interfaccia (nel namespace `testing`):

```
class CardinalityInterface {
 public:
  virtual ~CardinalityInterface();

  // Returns true if and only if call_count calls will satisfy this cardinality.
  virtual bool IsSatisfiedByCallCount(int call_count) const = 0;

  // Returns true if and only if call_count calls will saturate this
  // cardinality.
  virtual bool IsSaturatedByCallCount(int call_count) const = 0;

  // Describes self to an ostream.
  virtual void DescribeTo(std::ostream* os) const = 0;
};
```

Ad esempio, per specificare che una chiamata deve avvenire un numero pari di volte, è possibile scrivere

```
using ::testing::Cardinality;
using ::testing::CardinalityInterface;
using ::testing::MakeCardinality;

class EvenNumberCardinality : public CardinalityInterface {
 public:
  bool IsSatisfiedByCallCount(int call_count) const override {
    return (call_count % 2) == 0;
  }

  bool IsSaturatedByCallCount(int call_count) const override {
    return false;
  }

  void DescribeTo(std::ostream* os) const {
    *os << "called even number of times";
  }
};

Cardinality EvenNumber() {
  return MakeCardinality(new EvenNumberCardinality);
}

...
  EXPECT_CALL(foo, Bar(3))
      .Times(EvenNumber());
```

10.6.8 Scrivere Nuove Action {#QuickNewActions}

Se le azioni [action] native non funzionano, se ne può facilmente definirne una propria. Tutto ciò di cui si ha bisogno è un operatore di chiamata con una firma compatibile con la funzione mock-ata. Poi si può usare una lambda:

```
MockFunction<int(int)> mock;
EXPECT_CALL(mock, Call).WillOnce([](const int input) { return input * 7; });
EXPECT_EQ(mock.AsStdFunction()(2), 14);
```

O una struct con un operatore di chiamata (anche uno basato su template):

```
struct MultiplyBy {
  template <typename T>
  T operator()(T arg) { return arg * multiplier; }

  int multiplier;
};

// Then use:
// EXPECT_CALL(...).WillOnce(MultiplyBy{7});
```

Va bene anche che il callable non accetti argomenti, ignorando gli argomenti forniti alla funzione mock:

```
MockFunction<int(int)> mock;
EXPECT_CALL(mock, Call).WillOnce([] { return 17; });
EXPECT_EQ(mock.AsStdFunction()(0), 17);
```

Se utilizzato con `WillOnce`, il callable può presupporre che verrà chiamato al massimo una volta e può essere di tipo move-only:

```
// An action that contains move-only types and has an &&-qualified operator,
// demanding in the type system that it be called at most once. This can be
// used with WillOnce, but the compiler will reject it if handed to
// WillRepeatedly.
struct MoveOnlyAction {
  std::unique_ptr<int> move_only_state;
  std::unique_ptr<int> operator()() && { return std::move(move_only_state); }
};

MockFunction<std::unique_ptr<int>()> mock;
EXPECT_CALL(mock, Call).WillOnce(MoveOnlyAction{std::make_unique<int>(17)});
EXPECT_THAT(mock.AsStdFunction()(), Pointee(Eq(17)));
```

Più in generale, da utilizzare con una funzione mock la cui firma è `R(Args...)` loggetto può essere qualsiasi cosa convertibile in `OnceAction<R(Args...)>` o `Action<R(Args...)>`. La differenza tra i due è che `OnceAction` ha requisiti più deboli (`Action` richiede un input con costruttore-copia che può essere chiamato ripetutamente mentre `OnceAction` richiede un costruttore-move e supporta operatori di chiamata qualificati &&), ma può essere utilizzato solo con `WillOnce`. `OnceAction` è in genere rilevante solo quando si supportano tipi o azioni move-only che richiedono che un type-system garantisca che verranno chiamati al massimo una volta.

In genere non è necessario fare riferimento direttamente ai template `OnceAction` e `Action` nelle action: basta una struct o una classe con un operatore di chiamata, come negli esempi precedenti. Ma azioni polimorfiche più sofisticate che necessitano di conoscere il tipo di ritorno specifico della funzione mock possono definire operatori di conversione basati su template per renderlo possibile. Per gli esempi consultare `gmock-actions.h`.

Action Legacy basate su macro

Prima di C++11 le azioni basate su funtori non erano supportate; il vecchio modo di scriverle era attraverso una serie di macro `ACTION*`. Suggeriamo di evitarle nel nuovo codice; si nasconde molta logica dietro la macro, portando potenzialmente a errori del compilatore difficili da comprendere. Tuttavia, li descriviamo qui per completezza.

Scrivendo

```
ACTION(name) { statements; }
```

nello scope di un namespace (cioè non allinterno di una classe o funzione), si definirà unazione con il nome dato che esegue le istruzioni. Il valore restituito dalle `istruzioni` verrà utilizzato come valore di ritorno dellazione. Allinterno delle istruzioni, si può fare riferimento allargomento K-esimo (in base 0) della funzione mock come `argK`. Per esempio:

```
ACTION(IncrementArg1) { return ++(*arg1); }
```

consente di scrivere

```
... WillOnce(IncrementArg1());
```

Notare che non è necessario specificare i tipi degli argomenti della funzione mock. Sicuramente il codice è type-safe: si riceverà un errore del compilatore se `*arg1` non supporta loperatore ++, o se il tipo di `++(*arg1)` non è compatibile con il tipo restituito della funzione mock.

Un altro esempio:

```
ACTION(Foo) {
  (*arg2)(5);
  Blah();
  *arg1 = 0;
  return arg0;
}
```

definisce unazione `Foo()` che invoca largomento #2 (un puntatore a funzione) con 5, chiama la funzione `Blah()`, imposta il valore puntato dallargomento #1 a 0 e restituisce largomento #0.

Per maggiore comodità e flessibilità, si possono utilizzare i seguenti simboli predefiniti nel corpo di `ACTION`:

argK_type	Il tipo dellargomento K-esimo (in base 0) della funzione mock
args	Tutti gli argomenti della funzione mock come una tupla
args_type	Il tipo di tutti gli argomenti della funzione mock come tupla
return_type	Il tipo restituito della funzione mock
function_type	Il tipo della funzione mock

Ad esempio, quando si utilizza `ACTION` come azione stub per la funzione mock:

```
int DoSomething(bool flag, int* ptr);
```

abbiamo:

Simbolo Pre-definito	è [Bound] A
`arg0`	il valore di `flag`
`arg0_type`	il tipo `bool`
`arg1`	il valore di `ptr`
`arg1_type`	il tipo `int*`
`args`	la tuple `(flag, ptr)`
`args_type`	il tipo `std::tuple<bool, int*>`
`return_type`	il tipo `int`
`function_type`	il tipo `int(bool, int*)`

Azioni legacy parametrizzate basate su macro

Talvolta si vuole parametrizzare una action che si definisce. Per questo abbiamo unaltra macro

```
ACTION_P(name, param) { statements; }
```

Per esempio,

```
ACTION_P(Add, n) { return arg0 + n; }
```

permetterà di scrivere

```
// Returns argument #0 + 5.
... WillOnce(Add(5));
```

Per comodità, utilizziamo il termine *argomenti* per i valori utilizzati per invocare la funzione mock e il termine *parametri* per i valori utilizzati per istanziare unazione.

Notare che non è necessario fornire nemmeno il tipo del parametro. Supponiamo che il parametro si chiami `param`, si può anche utilizzare il simbolo definito da gMock `param_type` per fare riferimento al tipo del parametro come dedotto dal compilatore. Ad esempio, nel corpo di `ACTION_P(Add, n)` sopra, si può scrivere `n_type` per il tipo di `n`.

gMock fornisce anche `ACTION_P2`, `ACTION_P3` e così via per supportare azioni multi-parametro. Per esempio,

```
ACTION_P2(ReturnDistanceTo, x, y) {
  double dx = arg0 - x;
  double dy = arg1 - y;
  return sqrt(dx*dx + dy*dy);
}
```

permette di scrivere

```
... WillOnce(ReturnDistanceTo(5.0, 26.5));
```

Si può vedere `ACTION` come unazione parametrizzata degenerata in cui il numero di parametri è 0.

Si possono anche definire facilmente overload di azioni sul numero di parametri:

```
ACTION_P(Plus, a) { ... }
ACTION_P2(Plus, a, b) { ... }
```

10.6.9 Restrizioni sul tipo dellargomento o del parametro in una ACTION

Per la massima brevità e riusabilità, le macro `ACTION*` non chiedono di fornire i tipi degli argomenti della funzione mock né i parametri dellazione. Lasciamo invece che sia il compilatore a dedurre i tipi.

A volte, tuttavia, potremmo voler essere più espliciti riguardo ai tipi. Ci sono diversi trucchi per farlo. Per esempio:

```
ACTION(Foo) {
  // Makes sure arg0 can be converted to int.
  int n = arg0;
  ... use n instead of arg0 here ...
}

ACTION_P(Bar, param) {
  // Makes sure the type of arg1 is const char*.
  ::testing::StaticAssertTypeEq<const char*, arg1_type>();

  // Makes sure param can be converted to bool.
  bool flag = param;
}
```

dove `StaticAssertTypeEq` è unasserzione in fase di compilazione in googletest che verifica che due tipi siano uguali.

10.6.10 Scrivere Rapidamente una Nuove Template di Action

A volte è necessario fornire a unazione parametri template espliciti che non possono essere dedotti dai parametri del valore. `ACTION_TEMPLATE()` lo supporta e può essere visto come unestensione di `ACTION()` e di `ACTION_P*()`.

La sintassi:

```
ACTION_TEMPLATE(ActionName,
                HAS_m_TEMPLATE_PARAMS(kind1, name1, ..., kind_m, name_m),
                AND_n_VALUE_PARAMS(p1, ..., p_n)) { statements; }
```

definisce una action template che accetta *m* parametri template espliciti e *n* parametri di valore, dove *m* è in [1, 10] e *n* è in [0, 10]. `name_i` è il nome dell*i*-esimo parametro template, e `kind_i` specifica se si tratta di un `typename`, un integrale costante o un template. `p_i` è il nome del parametro valore *i*-esimo.

Esempio:

```
// DuplicateArg<k, T>(output) converts the k-th argument of the mock
// function to type T and copies it to *output.
ACTION_TEMPLATE(DuplicateArg,
                // Note the comma between int and k:
                HAS_2_TEMPLATE_PARAMS(int, k, typename, T),
                AND_1_VALUE_PARAMS(output)) {
  *output = T(std::get<k>(args));
}
```

Per creare unistanza di una action template, si scrive:

```
ActionName<t1, ..., t_m>(v1, ..., v_n)
```

dove le `t` sono gli argomenti template e le `v` sono gli argomenti valore. I tipi degli argomenti valore vengono dedotti dal compilatore. Per esempio:

```
using ::testing::_;
...
  int n;
  EXPECT_CALL(mock, Foo).WillOnce(DuplicateArg<1, unsigned char>(&n));
```

Per specificare esplicitamente i tipi degli argomenti valore, si possono fornire argomenti template aggiuntivi:

```
ActionName<t1, ..., t_m, u1, ..., u_k>(v1, ..., v_n)
```

dove `u_i` è il tipo desiderato di `v_i`.

`ACTION_TEMPLATE` e `ACTION/ACTION_P*` si possono overload-are sul numero di parametri valore, ma non sul numero di parametri template. Senza la restrizione, il significato di quanto segue non è chiaro:

```
  OverloadedAction<int, bool>(x);
```

Stiamo utilizzando unazione con parametro tempale singolo in cui `bool` si riferisce al tipo di `x`, o unazione con due parametri template in cui al compilatore viene chiesto di dedurre il tipo di `x`?

10.6.11 Usare il Tipo dellOggetto ACTION

Per scrivere una funzione che restituisce un oggetto `ACTION`, se ne deve conoscere il tipo. Il tipo dipende dalla macro utilizzata per definire lazione e i tipi dei parametri. La regola è relativamente semplice:

Definizione Data	Espressione	Ha il Tipo
`ACTION(Foo)`	`Foo()`	`FooAction`
`ACTION_TEMPLATE(Foo, HAS_m_TEMPLATE_PARAMS(...), AND_0_VALUE_PARAMS())`	`Foo<t1, ..., t_m>()`	`FooAction<t1, ..., t_m>`
`ACTION_P(Bar, param)`	`Bar(int_value)`	`BarActionP<int>`
`ACTION_TEMPLATE(Bar, HAS_m_TEMPLATE_PARAMS(...), AND_1_VALUE_PARAMS(p1))`	`Bar<t1, ..., t_m>(int_value)`	`BarActionP<t1, ..., t_m, int>`
`ACTION_P2(Baz, p1, p2)`	`Baz(bool_value, int_value)`	`BazActionP2<bool, int>`
`ACTION_TEMPLATE(Baz, HAS_m_TEMPLATE_PARAMS(...), AND_2_VALUE_PARAMS(p1, p2))`	`Baz<t1, ..., t_m>(bool_value, int_value)`	`BazActionP2<t1, ..., t_m, bool, int>`

Notare che dobbiamo scegliere suffissi diversi (`Action`, `ActionP`, `ActionP2`, ecc.) per azioni con numeri diversi di parametri valore, oppure non si possono avere overload delle definizioni delle azioni rispetto al loro numero.

10.6.12 Scrivere Nuove Azioni Monomorfiche {#NewMonoActions}

Sebbene le macro `ACTION*` siano molto comode, a volte sono inappropriate. Ad esempio, nonostante i trucchi mostrati nelle ricette precedenti, non consentono di specificare direttamente i tipi degli argomenti della funzione mock e dei parametri dellazione, il che in generale porta a messaggi di errore del compilatore non ottimizzati che possono confondere gli utenti non esperti. Inoltre, non consentono overload di azioni basate sui tipi di parametri senza eseguire alcuni passaggi.

Unalternativa alle macro `ACTION*` è implementare `::testing::ActionInterface<F>`, dove `F` è il tipo della funzione mock in cui verrà utilizzata lazione. Per esempio:

```
template <typename F>
class ActionInterface {
 public:
  virtual ~ActionInterface();

  // Performs the action.  Result is the return type of function type
  // F, and ArgumentTuple is the tuple of arguments of F.
  //

  // For example, if F is int(bool, const string&), then Result would
  // be int, and ArgumentTuple would be std::tuple<bool, const string&>.
  virtual Result Perform(const ArgumentTuple& args) = 0;
};
```

```
using ::testing::_;
using ::testing::Action;
using ::testing::ActionInterface;
using ::testing::MakeAction;

typedef int IncrementMethod(int*);

class IncrementArgumentAction : public ActionInterface<IncrementMethod> {
 public:
  int Perform(const std::tuple<int*>& args) override {
    int* p = std::get<0>(args);  // Grabs the first argument.
    return *p++;
  }
};

Action<IncrementMethod> IncrementArgument() {
  return MakeAction(new IncrementArgumentAction);
}

...
  EXPECT_CALL(foo, Baz(_))
      .WillOnce(IncrementArgument());

  int n = 5;
  foo.Baz(&n);  // Should return 5 and change n to 6.
```

10.6.13 Scrivere Nuove Action Polimorfiche {#NewPolyActions}

La ricetta precedente ha mostrato come definire una propria azione. Va tutto bene, tranne che è necessario conoscere il tipo della funzione in cui verrà utilizzata lazione. A volte questo può essere un problema. Ad esempio, per utilizzare lazione in funzioni con tipi *diversi* (ad esempio come `Return()` e `SetArgPointee()`).

Se unazione può essere utilizzata in diversi tipi di funzioni mock, diciamo che è *polimorfica*. La funzione template `MakePolymorphicAction()` semplifica la definizione di tale azione:

```
namespace testing {
template <typename Impl>
PolymorphicAction<Impl> MakePolymorphicAction(const Impl& impl);
}  // namespace testing
```

Come esempio, definiamo unazione che restituisce il secondo argomento nellelenco degli argomenti della funzione mock. Il primo passo è definire una classe di implementazione:

```
class ReturnSecondArgumentAction {
 public:
  template <typename Result, typename ArgumentTuple>
  Result Perform(const ArgumentTuple& args) const {
    // To get the i-th (0-based) argument, use std::get(args).
    return std::get<1>(args);
  }
};
```

Questa classe di implementazione *non* ha bisogno di ereditare da una classe particolare. Ciò che conta è che deve avere un metodo template `Perform()`. Questo metodo template accetta gli argomenti della funzione mock come una tupla in un argomento **singolo** e restituisce il risultato dellazione. Può essere `const` o meno, ma deve essere invocabile esattamente con un argomento template, che è il tipo del risultato. In altre parole, si deve essere in grado di chiamare `Perform<R>(args)` dove `R` è il tipo restituito dalla funzione mock e `args` sono i suoi argomenti in una tupla.

Poi, usiamo `MakePolymorphicAction()` per trasformare unistanza della classe di implementazione nellazione polimorfica di cui abbiamo bisogno. Sarà conveniente avere un wrapper per questo:

```
using ::testing::MakePolymorphicAction;
using ::testing::PolymorphicAction;

PolymorphicAction<ReturnSecondArgumentAction> ReturnSecondArgument() {
  return MakePolymorphicAction(ReturnSecondArgumentAction());
}
```

Ora si può utilizzare questa azione polimorfica nello stesso modo in cui si usano quelle native:

```
using ::testing::_;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(int, DoThis, (bool flag, int n), (override));
  MOCK_METHOD(string, DoThat, (int x, const char* str1, const char* str2),
              (override));
};

  ...
  MockFoo foo;
  EXPECT_CALL(foo, DoThis).WillOnce(ReturnSecondArgument());
  EXPECT_CALL(foo, DoThat).WillOnce(ReturnSecondArgument());
  ...
  foo.DoThis(true, 5);  // Will return 5.
  foo.DoThat(1, "Hi", "Bye");  // Will return "Hi".
```

10.6.14 Insegnare a gMock Come Stampare i Propri Valori

Quando avviene una chiamata [uninteresting] o inaspettata [unexpected], gMock stampa i valori degli argomenti e lo stack trace come aiuto al debug. Anche le macro di asserzione come `EXPECT_THAT` e `EXPECT_EQ` stampano i valori in questione quando lasserzione fallisce. gMock e googletest lo fanno utilizzando la stampante di valori estensibile dallutente di googletest.

Questa stampante sa lavorare con i tipi nativi di C++, gli array, i contenitori STL e qualsiasi tipo che supporti loperatore «. Per gli altri tipi, stampa i byte grezzi nel valore e spera che lutente possa capirlo. La Guida Avanzata di GoogleTest spiega come estendere la stampante per eseguire un lavoro migliore stampando il tipo particolare piuttosto che il dump dei byte.

10.7 Mock Utili Creati Con gMock

10.7.1 Mock std::function {#MockFunction}

`std::function` è un tipo di funzione generale introdotto in C++11. È un modo preferito per passare le callback a nuove interfacce. Le funzioni sono copiabili e di solito non vengono passate tramite puntatore, il che le rende difficili da rendere mock. Niente paura - `MockFunction` è daiuto.

`MockFunction<R(T1, ..., Tn)>` ha un metodo mock `Call()` con la signature:

```
  R Call(T1, ..., Tn);
```

Cè anche il metodo `AsStdFunction()`, che crea un proxy `std::function` che inoltra [forward] a Call:

```
  std::function<R(T1, ..., Tn)> AsStdFunction();
```

Per utilizzare `MockFunction`, si crea prima un oggetto `MockFunction` e si impostano le expectation sul suo metodo `Call`. Poi si passa il proxy ottenuto da `AsStdFunction()` al codice da testare. Per esempio:

```
TEST(FooTest, RunsCallbackWithBarArgument) {
  // 1. Create a mock object.
  MockFunction<int(string)> mock_function;

  // 2. Set expectations on Call() method.
  EXPECT_CALL(mock_function, Call("bar")).WillOnce(Return(1));

  // 3. Exercise code that uses std::function.
  Foo(mock_function.AsStdFunction());
  // Foo's signature can be either of:
  // void Foo(const std::function<int(string)>& fun);
  // void Foo(std::function<int(string)> fun);

  // 4. All expectations will be verified when mock_function
  //     goes out of scope and is destroyed.
}
```

Da ricordare che gli oggetti funzione creati con `AsStdFunction()` vengono solo [forwarder]. Se se ne creano di più, condivideranno lo stesso insieme di expectation.

Sebbene `std::function` supporti un numero illimitato di argomenti, limplementazione di `MockFunction` è limitata a dieci. Se mai si raggiungesse questo limite beh, la richiamata ha problemi più grandi del poter essere mock. :-)

CAPITOLO 11

Cheat Sheet di gMock

11.1 Definizione di una Classe Mock

11.1.1 Mock di una Classe Normale {#MockClass}

Dato

```
class Foo {
 public:
  virtual ~Foo();
  virtual int GetSize() const = 0;
  virtual string Describe(const char* name) = 0;
  virtual string Describe(int type) = 0;
  virtual bool Process(Bar elem, int count) = 0;
};
```

(notare che `~Foo()` **deve** essere virtual) possiamo definirne il mock come

```
#include <gmock/gmock.h>

class MockFoo : public Foo {
 public:
  MOCK_METHOD(int, GetSize, (), (const, override));
  MOCK_METHOD(string, Describe, (const char* name), (override));
  MOCK_METHOD(string, Describe, (int type), (override));
  MOCK_METHOD(bool, Process, (Bar elem, int count), (override));
};
```

Per creare un ńniceż mock, che ignora tutte le chiamate [uninteresting], un mock ńnaggyż, che emette warning su tutte le chiamate [uninteresting], o un mock ństrictż, che li tratta come fallimenti:

```
using ::testing::NiceMock;
using ::testing::NaggyMock;
using ::testing::StrictMock;

NiceMock<MockFoo> nice_foo;      // The type is a subclass of MockFoo.
```

(continues on next page)

(continua dalla pagina precedente)

```
NaggyMock<MockFoo> naggy_foo;    // The type is a subclass of MockFoo.
StrictMock<MockFoo> strict_foo;  // The type is a subclass of MockFoo.
```

{: .callout .note} **Nota:** Un oggetto mock è attualmente naggy per default. Potremmo renderlo nice per default in futuro.

11.1.2 Mock di una Classe Template {#MockTemplate}

Le classi templates si possono rendere mocked proprio come qualsiasi classe.

Per il mock di

```
template <typename Elem>
class StackInterface {
 public:
  virtual ~StackInterface();
  virtual int GetSize() const = 0;
  virtual void Push(const Elem& x) = 0;
};
```

(notare che il mock di tutte le funzioni membro, compreso `~StackInterface()` **devono** essere virtual).

```
template <typename Elem>
class MockStack : public StackInterface<Elem> {
 public:
  MOCK_METHOD(int, GetSize, (), (const, override));
  MOCK_METHOD(void, Push, (const Elem& x), (override));
};
```

11.1.3 Specificare le Convenzioni di Chiamata per le Funzioni Mock

Se la funzione mock non utilizza la convenzione di chiamata di default, la si può specificare aggiungendo `Calltype(convention)` al quarto parametro di `MOCK_METHOD`. Per esempio,

```
  MOCK_METHOD(bool, Foo, (int n), (Calltype(STDMETHODCALLTYPE)));
  MOCK_METHOD(int, Bar, (double x, double y),
              (const, Calltype(STDMETHODCALLTYPE)));
```

dove `STDMETHODCALLTYPE` è definito da `<objbase.h>` su Windows.

11.2 Uso dei Mock nei Test {#UsingMocks}

Il flusso di lavoro tipico è:

1. Si importano i nomi gMock da utilizzare. Tutti i simboli gMock si trovano nel namespace `testing` a meno che non siano macro o diversamente indicati.
2. Si creano gli oggetti mock.
3. Facoltativamente, imposta le azioni di default degli oggetti mock.
4. Si settano le expectation sugli oggetti mock (Come verranno chiamati? Cosa faranno?).
5. Codice di esercizio che utilizza gli oggetti mock; se necessario, si controlla il risultato utilizzando le asserzioni di googletest.
6. Quando un oggetto mock viene distrutto, gMock verifica automaticamente che tutte le expectation su di esso siano state soddisfatte.

Ecco un esempio:

```
using ::testing::Return;                          // #1

TEST(BarTest, DoesThis) {
  MockFoo foo;                                    // #2

  ON_CALL(foo, GetSize())                         // #3
      .WillByDefault(Return(1));
  // ... other default actions ...

  EXPECT_CALL(foo, Describe(5))                   // #4
      .Times(3)
      .WillRepeatedly(Return("Category 5"));
  // ... other expectations ...

  EXPECT_EQ(MyProductionFunction(&foo), "good");  // #5
}                                                 // #6
```

11.3 Impostazione delle Azioni di Default {#OnCall}

gMock ha un**azione di default nativa** per qualsiasi funzione che restituisce `void`, `bool`, un valore numerico o un puntatore. In C++11, it additionally returns the default-constructed value, if one exists for the given type.

Per personalizzare lazione di default per le funzioni con tipo restituito `T`, si usa *`DefaultValue<T>`*. Per esempio:

```
  // Sets the default action for return type std::unique_ptr<Buzz> to
  // creating a new Buzz every time.
  DefaultValue<std::unique_ptr<Buzz>>::SetFactory(
      [] { return std::make_unique<Buzz>(AccessLevel::kInternal); });

  // When this fires, the default action of MakeBuzz() will run, which
  // will return a new Buzz object.
  EXPECT_CALL(mock_buzzer_, MakeBuzz("hello")).Times(AnyNumber());

  auto buzz1 = mock_buzzer_.MakeBuzz("hello");
  auto buzz2 = mock_buzzer_.MakeBuzz("hello");
  EXPECT_NE(buzz1, nullptr);
  EXPECT_NE(buzz2, nullptr);
  EXPECT_NE(buzz1, buzz2);

  // Resets the default action for return type std::unique_ptr<Buzz>,
  // to avoid interfere with other tests.
  DefaultValue<std::unique_ptr<Buzz>>::Clear();
```

Per personalizzare lazione di default per un metodo particolare di uno specifico oggetto mock, si usa *`ON_CALL`*. `ON_CALL` ha una sintassi simile a `EXPECT_CALL`, ma viene utilizzato per impostare comportamenti di default quando non è necessario che venga chiamato il metodo mock. Vedere Quando usare Expect per una discussione più dettagliata.

11.4 Impostare le Expectation {#ExpectCall}

Vedere *EXPECT_CALL* nella ńGuida al Mockingż.

11.5 I Matcher {#MatcherList}

Vedere i *Riferimenti ai Matcher*.

11.6 Le Action {#ActionList}

Vedere i *Riferimenti alle Action*.

11.7 Cardinalità {#CardinalityList}

Vedere le *clausole* `Times` di `EXPECT_CALL` nella ńGuida al Mockingż.

11.8 Ordine delle Expectation

Per default, le expectation possono essere soddisfatte in *qualsiasi* ordine. Se alcune o tutte le expectation devono essere soddisfatte in un determinato ordine, si può utilizzare la *clausola* `After` o la *clausola* `InSequence` di `EXPECT_CALL`, oppure usare un *oggetto* `InSequence`.

11.9 Verificare e Resettare un Mock

gMock verificherà le expectation su un oggetto mock quando viene distrutto, oppure lo si può fare prima:

```
using ::testing::Mock;
...
// Verifies and removes the expectations on mock_obj;
// returns true if and only if successful.
Mock::VerifyAndClearExpectations(&mock_obj);
...
// Verifies and removes the expectations on mock_obj;
// also removes the default actions set by ON_CALL();
// returns true if and only if successful.
Mock::VerifyAndClear(&mock_obj);
```

Non stabilire nuove expectation dopo aver verificato e ripulito un mock dopo il suo utilizzo. Limpostazione delle expectation dopo il codice che esercita il mock ha un comportamento indefinito. Vedere Uso dei Mock nei Test per ulteriori informazioni.

Si può anche dire a gMock che un oggetto mock può essere perso [leaked] e non è necessario verificarlo:

```
Mock::AllowLeak(&mock_obj);
```

11.10 Classi Mock

gMock definisce un comodo template di classe mock

```
class MockFunction<R(A1, ..., An)> {
 public:
  MOCK_METHOD(R, Call, (A1, ..., An));
};
```

Consultare questa ricetta per una sua applicazione.

11.11 I Flag

Flag	Descrizione
`--gmock_catch_leaked_mo`	Non segnalare gli oggetti mock [leaked] come fallimenti.
`--gmock_verbose=LEVEL`	Imposta il livello di verbosità di default (`info`, `warning`, o `error`) dei messaggi di Google Mock.

CAPITOLO 12

FAQ di GoogleTest

12.1 Perché i nomi delle test suite e dei test non devono contenere un underscore?

{: .callout .note} Nota: GoogleTest si riserva lunderscore (_) per parole chiave con scopi speciali, come il prefisso `DISABLED_`, in aggiunta alla motivazione seguente.

Underscore (_) è speciale, poiché il C++ riserva quanto segue per essere utilizzato dal compilatore e dalla libreria standard:

1. qualsiasi identificatore che inizia con un _ seguito da una lettera maiuscola e
2. qualsiasi identificatore che contiene due underscore consecutivi (cioè __) *ovunque* nel suo nome.

Al codice utente è *vietato* utilizzare tali identificatori.

Ora vediamo cosa significa per `TEST` e `TEST_F`.

Attualmente `TEST(TestSuiteName, TestName)` genera una classe denominata `TestSuiteName_TestName_Test`. Cosa succede se `TestSuiteName` o `TestName` contengono _?

1. Se `TestSuiteName` inizia con un _ seguito da una lettera maiuscola (ad esempio `_Foo`), finiamo con `_Foo_TestName_Test`, che è riservato e quindi non valido.
2. Se `TestSuiteName` termina con un _ (ad esempio `Foo_`), otteniamo `Foo__TestName_Test`, che non è valido.
3. Se `TestName` inizia con un _ (ad esempio `_Bar`), otteniamo `TestSuiteName__Bar_Test`, che non è valido.
4. Se `TestName` termina con un _ (ad esempio `Bar_`), otteniamo `TestSuiteName_Bar__Test`, che non è valido.

Quindi chiaramente `TestSuiteName` e `TestName` non possono iniziare o finire con _ (In realtà, `TestSuiteName` può iniziare con _purché il _ non sia seguito da una lettera maiuscola. Ma la cosa sta diventando complicata. Quindi per semplicità diciamo che non può iniziare con _).

Potrebbe sembrare corretto che `TestSuiteName` e `TestName` contengano _ nel mezzo. Tuttavia, cè questo da considerare:

```
TEST(Time, Flies_Like_An_Arrow) { ... }
TEST(Time_Flies, Like_An_Arrow) { ... }
```

Ora, i due TEST genereranno entrambi la stessa classe (Time_Flies_Like_An_Arrow_Test). Questo non è buono.

Quindi, per semplicità, chiediamo agli utenti di evitare _ in TestSuiteName e in TestName. La regola è più vincolante del necessario, ma è semplice e facile da ricordare. Offre inoltre a GoogleTest un po di margine di manovra nel caso in cui la sua implementazione debba cambiare in futuro.

Violando la regola, potrebbero non esserci conseguenze immediate, ma il test potrebbe (solo potrebbe) ńrompersiż con un nuovo compilatore (o una nuova versione del compilatore utilizzato) o con una nuova versione di GoogleTest. Quindi è meglio seguire la regola.

12.2 Perché GoogleTest supporta EXPECT_EQ(NULL, ptr) e ASSERT_EQ(NULL, ptr) ma non EXPECT_NE(NULL, ptr) e ASSERT_NE(NULL, ptr)?

Prima di tutto, si può usare nullptr con ciascuna di queste macro, ad es. EXPECT_EQ(ptr, nullptr), EXPECT_NE(ptr, nullptr), ASSERT_EQ(ptr, nullptr), ASSERT_NE(ptr, nullptr). Questa è la sintassi preferita nella guida stilistica perché nullptr non presenta i problemi sul tipo che presenta NULL.

A causa di alcune peculiarità del C++, sono necessari alcuni trucchi di metaprogrammazione non banali per supportare luso di NULL come argomento delle macro EXPECT_XX() e ASSERT_XX(). Pertanto lo facciamo solo dove è più necessario (altrimenti rendiamo limplementazione di GoogleTest più difficile da mantenere e più soggetta a errori del necessario).

Storicamente, la macro EXPECT_EQ() prendeva il valore *expected* come primo argomento e il valore *actual* come secondo, sebbene lordine degli argomenti ora sia scoraggiato. Era ragionevole che qualcuno volesse scrivere EXPECT_EQ(NULL, some_expression), e in effetti questo è stato richiesto più volte. Pertanto lo abbiamo implementato.

La necessità di EXPECT_NE(NULL, ptr) non era così pressante. Quando lasserzione fallisce, si sa già che ptr deve essere NULL, quindi non aggiunge alcuna informazione per stampare ptr in questo caso. Ciò significa che EXPECT_TRUE(ptr != NULL) funziona altrettanto bene.

Se dovessimo supportare EXPECT_NE(NULL, ptr), per coerenza dovremmo supportare anche EXPECT_NE(ptr, NULL). Ciò significa utilizzare i trucchi della metaprogrammazione del template due volte nellimplementazione, rendendolo ancora più difficile da comprendere e mantenere. Riteniamo che il vantaggio non giustifichi il costo.

Infine, con la crescita della libreria di matcher gMock, stiamo incoraggiando le persone a utilizzare la sintassi unificata EXPECT_THAT(value, matcher) più spesso nei test. Un vantaggio significativo dellapproccio con i matcher è che questi possono essere facilmente combinati per formare nuovi matcher, mentre le macro EXPECT_NE, ecc, non possono essere facilmente combinate. Pertanto vogliamo investire di più nei matcher che nelle macro EXPECT_XX().

12.3 Devo verificare che diverse implementazioni di uninterfaccia soddisfino alcuni requisiti comuni. Dovrei utilizzare test tipizzati o test con valori parametrizzati?

Per testare varie implementazioni della stessa interfaccia, è possibile eseguire test tipizzati o test con valori parametrizzati. È davvero personale decidere quale sia più conveniente, a seconda del caso particolare. Alcune linee guida approssimative:

- I test tipizzati possono essere più facili da scrivere se le istanze delle diverse implementazioni possono essere create allo stesso modo, modulo il tipo. Ad esempio, se tutte queste implementazioni hanno un costruttore pubblico di default (in modo che sia possibile scrivere new TypeParam), o se le loro funzioni factory hanno la stessa forma (ad esempio CreateInstance<TypeParam>()).
- I test con valori parametrizzati possono essere più facili da scrivere se sono necessari pattern di codice diversi per creare istanze di implementazioni diverse, ad es. new Foo contro new Bar(5). Per compensare

le differenze, è possibile scrivere wrapper di funzioni factory e passare questi puntatori a funzione ai test come parametri.

- Quando un test tipizzato fallisce, loutput di default include il nome del tipo, che può aiutare a identificare rapidamente quale implementazione sia sbagliata. Per default, i test con valori parametrizzati mostrano solo il numero delliterazione non riuscita. Sarà necessario definire una funzione che restituisca il nome delliterazione e passarla come terzo parametro a INSTANTIATE_TEST_SUITE_P per avere un output più utile.
- Quando si usano test tipizzati, si deve eseguire il test rispetto al tipo di interfaccia, non ai tipi concreti (in altre parole, `implicit_cast<MyInterface*>(my_concrete_impl)` deve funzionare, non basta che funzioni `my_concrete_impl`). È meno probabile che si commettano errori in questarea quando si utilizzano test con valori parametrizzati.

Speriamo di non aver aumentato la confusione. :-) Se possibile, suggeriamo di provare entrambi gli approcci. La pratica è un modo molto migliore per cogliere le sottili differenze tra i due strumenti. Una volta acquisita unesperienza concreta, si potrà decidere molto più facilmente quale utilizzare.

12.4 Il mio ńdeath testż modifica alcuni stati, ma il cambiamento sembra perso al termine del test. Perché?

I death test (`EXPECT_DEATH`, ecc.) vengono eseguiti in un sottoprocesso s.t. il crash previsto non kill-erà il programma di test (ovvero il processo genitore). Di conseguenza, eventuali effetti collaterali in memoria che avvengono sono osservabili nei rispettivi sottoprocessi, ma non nel processo genitore. Si possono pensare come se venissero eseguiti in un universo parallelo, più o meno.

In particolare, se si utilizza il mocking e lenunciato del death test invoca alcuni metodi mock, il processo genitore penserà che le chiamate non siano mai avvenute. Pertanto, si potrebbero spostare le istruzioni `EXPECT_CALL` nella macro `EXPECT_DEATH`.

12.5 EXPECT_EQ(htonl(blah), blah_blah) genera strani errori del compilatore in modalità opt. È un bug di GoogleTest?

In realtà il bug è in `htonl()`.

Secondo `'man htonl'`, `htonl()` è una *funzione*, il che significa che è corretto utilizzare `htonl` come puntatore a funzione. Tuttavia, in modalità opt `htonl()` è definito come una *macro*, il che interrompe questo utilizzo.

Peggio ancora, la definizione della macro di `htonl()` utilizza unestensione di `gcc` che *non* è C++ standard. Questa implementazione complicata presenta alcune limitazioni ad hoc. In particolare, impedisce di scrivere `Foo<sizeof(htonl(x))>()`, dove `Foo` è un template che ha un argomento intero.

Limplementazione di `EXPECT_EQ(a, b)` usa `sizeof(... a ...)` allinterno di un argomento template e quindi non viene compilato in modalità opt quando `a` contiene una chiamata a `htonl()`. È difficile fare in modo che `EXPECT_EQ` bypassi il bug `htonl()`, poiché la soluzione deve funzionare con diversi compilatori su varie piattaforme.

12.6 Il compilatore si lamenta di ńriferimenti non definitiż ad alcune variabili membro const statiche, ma sono state definite nel corpo della classe. Cosa cè che non va?

Se la classe ha un membro dati static:

```
// foo.h
class Foo {
  ...
```

(continues on next page)

(continua dalla pagina precedente)

```
  static const int kBar = 100;
};
```

lo si deve definire anche *al di fuori* del corpo della classe in `foo.cc`:

```
const int Foo::kBar;  // No initializer here.
```

Altrimenti il codice sarà **C++ invalido**, e potrebbe rompersi in modi imprevisti. In particolare, utilizzarlo nelle asserzioni di confronto di GoogleTest (`EXPECT_EQ`, ecc.) genererà un errore del linker ńundefined referenceż. Il fatto che ńfunzionavaż non significa che sia valido. Significa solo che si è stati fortunati. :-)

Se la dichiarazione del membro dati statico è `constexpr` allora è implicitamente una definizione `inline` e non è necessaria una definizione separata in `foo.cc`:

```
// foo.h
class Foo {
  ...
  static constexpr int kBar = 100;  // Defines kBar, no need to do it in foo.cc.
};
```

12.7 Posso derivare una fixture da unaltra?

Sì.

Ogni fixture ha una test suite corrispondente e con lo stesso nome. Ciò significa che solo una test suite può utilizzare una particolare fixture. A volte, tuttavia, più casi di test potrebbero voler utilizzare le stesse fixture o leggermente diverse. Ad esempio, si potrebbe vole essere sicuri che tutte le test suite di una libreria GUI non abbiano importanti ńleakż di risorse di sistema come font e brush.

In GoogleTest, si condivide una fixture tra le test suite inserendo la logica condivisa in una fixture di base, poi derivando da quella base una fixture separata per ciascuna test suite che voglia utilizzare questa logica comune. Poi si usa `TEST_F()` per scrivere i test utilizzando ciascuna fixture derivata.

In genere, il codice è simile al seguente:

```
// Defines a base test fixture.
class BaseTest : public ::testing::Test {
 protected:
  ...
};

// Derives a fixture FooTest from BaseTest.
class FooTest : public BaseTest {
 protected:
  void SetUp() override {
    BaseTest::SetUp();  // Sets up the base fixture first.
    ... additional set-up work ...
  }

  void TearDown() override {
    ... clean-up work for FooTest ...
    BaseTest::TearDown();  // Remember to tear down the base fixture
                           // after cleaning up FooTest!
  }

  ... functions and variables for FooTest ...
};
```

(continues on next page)

(continua dalla pagina precedente)

```
// Tests that use the fixture FooTest.
TEST_F(FooTest, Bar) { ... }
TEST_F(FooTest, Baz) { ... }

... additional fixtures derived from BaseTest ...
```

Se necessario, è possibile continuare a derivare fixture da una fixture derivata. GoogleTest non ha limiti sulla profondità della gerarchia.

Per un esempio completo sulluso delle test fixture derivate, vedere sample5_unittest.cc.

12.8 Il mio compilatore dice ńvoid value not ignored as it ought to beż. Cosa significa?

Probabilmente si sta utilizzando un `ASSERT_*()` in una funzione che non restituisce `void`. `ASSERT_*()` può essere utilizzato solo nelle funzioni `void`, poiché le eccezioni sono disabilitate dal nostro sistema di build. Consultare ulteriori dettagli qui.

12.9 Il mio death test si blocca (o ci sono errori seg-fault). Come lo si ripara?

In GoogleTest, i death test vengono eseguiti in un processo figlio e il modo in cui funzionano è delicato. Per scrivere dei death test si devono veramente capire come funzionano: vederei i dettagli nelle *Asserzioni Death* nei Riferimenti sulle Asserzioni.

In particolare, ai death test non piace avere più thread nel processo genitore. Quindi la prima cosa che si può provare è eliminare la creazione di thread al di fuori di `EXPECT_DEATH()`. Ad esempio, si potrebbero utilizzare oggetti mock o fake invece di quelli reali nei test.

A volte questo è impossibile poiché alcune librerie da utilizzare potrebbero creare thread prima ancora che venga raggiunto il `main()`. In questo caso, si può provare a ridurre al minimo la possibilità di conflitti spostando quante più attività possibili allinterno di `EXPECT_DEATH()` (nel caso estremo, si può spostare tutto allinterno), o lasciandone poche cose. Inoltre, si può provare a impostare lo stile del death test su `"threadsafe"`, che è più sicuro ma più lento, e vedere se aiuta.

Se si usano i death test thread-safe, ricordarsi che eseguono nuovamente il programma di test dallinizio nel processo figlio. Il programma deve poter funzionare fianco a fianco con se stesso e che sia deterministico.

Alla fine, questo si riduce a una buona programmazione concorrente. Non ci devono essere condizioni di race o deadlock nel programma. Spiacenti: nessuna soluzione miracolosa!

12.10 Si usa il costruttore/distruttore della fixture o SetUp()/TearDown()? {#CtorVsSetUp}

La prima cosa da ricordare è che that GoogleTest **non** riutilizza lo stesso oggetto test fixture in più test. Per ciascun `TEST_F`, GoogleTest creerà un **nuovo** oggetto fixture, chiamerà immediatamente `SetUp()`, eseguirà il corpo del test, chiamerà `TearDown()` per poi eseguire il delete delloggetto fixture test.

Quando è necessario scrivere la logica di configurazione e ripulitura per ciascun test, è possibile scegliere tra utilizzare il costruttore/distruttore della fixture o `SetUp()`/`TearDown()`. Il primo è solitamente preferito perché presenta i seguenti vantaggi:

- Inizializzando una variabile membro nel costruttore, abbiamo la possibilità di renderla `const`, il che aiuta a prevenire modifiche accidentali al suo valore e rende i test ovviamente più corretti.
- Nel caso in cui abbiamo bisogno di sottoclassi della classe fixture, è garantito che il costruttore della sottoclasse chiamerà *prima* quello della classe base e il distruttore della sottoclasse è garantito che chiamerà il

distruttore della classe base *dopo*. Con `SetUp()/TearDown()`, una sottoclasse potrebbe commettere lerrore di dimenticare di chiamare la classe base di `SetUp()/TearDown()` o di chiamarla nel momento sbagliato.

Si potrebbe comunque usare `SetUp()/TearDown()` nei seguenti casi:

- Il C++ non consente chiamate di funzioni virtuali nei costruttori e nei distruttori. Si può chiamare un metodo dichiarato come virtuale, ma non utilizzerà il dispatch dinamico. Utilizzerà la definizione della classe il cui costruttore è attualmente in esecuzione. Questo perché chiamare un metodo virtuale prima che il costruttore della classe derivata abbia la possibilità di essere eseguito è molto pericoloso: il metodo virtuale potrebbe operare su dati non inizializzati. Pertanto, dovendo chiamare un metodo che verrà sovrascritto in una classe derivata, si deve usare `SetUp()/TearDown()`.
- Nel corpo di un costruttore (o di un distruttore), non è possibile utilizzare le macro `ASSERT_xx`. Pertanto, se loperazione di configurazione potrebbe causare un fallimento fatale del test che dovrebbe impedire lesecuzione del test, è necessario utilizzare `abort` e abortire lintero eseguibile del test, oppure utilizzare `SetUp()` invece di un costruttore.
- Se loperazione di ripulitura (tear-down) può generare uneccezione, si deve usare `TearDown()` invece del distruttore, poiché linserimento di un distruttore porta a un comportamento indefinito e solitamente ńkilleràż immediatamente il programma. Notare che molte librerie standard (come STL) potrebbero generare uneccezione quando queste sono abilitate nel compilatore. Pertanto è preferibile `TearDown()` se si vogliono scrivere test portabili che funzionino con o senza eccezioni.
- Il team di GoogleTest sta valutando la possibilità di lanciare eccezioni [throw] nelle macro di asserzione su piattaforme in cui sono abilitate le eccezioni (ad esempio Windows, Mac OS e Linux lato client), il che eliminerà la necessità per lutente di propagare gli errori da una subroutine al suo chiamante. Pertanto, non si dovrebbero usare le asserzioni GoogleTest in un distruttore se il codice può essere eseguito su tali piattaforme.

12.11 Il compilatore dice ńno matching function to callż quando si usa `ASSERT_PRED*`. Come lo si ripara?

Vedere i dettagli per *`EXPECT_PRED*`* nei ńRiferimenti sulle Asserzioniż.

12.12 Il compilatore dice ńignoring return valueż quando si chiama RUN_ALL_TESTS(). Perché?

Qualcuno ha ignorato il valore restituito da `RUN_ALL_TESTS()`. Cioè, invece di

```
return RUN_ALL_TESTS();
```

scrivono

```
RUN_ALL_TESTS();
```

Questo è **sbagliato e pericoloso**. I servizi di test devono vedere il valore restituito di `RUN_ALL_TESTS()` per determinare se un test è stato superato. Se la funzione `main()` lo ignora, il test verrà considerato riuscito anche se presenta un errore nellasserzione di GoogleTest. Molto brutto.

Abbiamo deciso di risolvere questo problema (grazie a Michael Chastain per lidea). Ora il codice non sarà più in grado di ignorare `RUN_ALL_TESTS()` quando verrà compilato con `gcc`. Facendolo, si riceverà un errore del compilatore.

Se il compilatore si lamenta del fatto che viene ignorato il valore restituito di `RUN_ALL_TESTS()`, la soluzione è semplice: si deve utilizzare il valore come valore restituito da `main()`.

Ma come potremmo introdurre un cambiamento che renda non funzionanti i test esistenti? Bene, in questo caso il codice era già bacato, quindi non è stato guastato adesso. :-)

12.13 Il compilatore dice che un costruttore (o un distruttore) non può restituire un valore. Cosa sta succedendo?

A causa di una peculiarità del C++, per supportare la sintassi per lo streaming dei messaggi su un `ASSERT_*`, ad es.

```
  ASSERT_EQ(1, Foo()) << "blah blah" << foo;
```

abbiamo dovuto rinunciare a utilizzare `ASSERT*` e `FAIL*` (ma non `EXPECT*` e `ADD_FAILURE*`) nei costruttori e nei distruttori. La soluzione alternativa è quella di spostare il contenuto del costruttore/distruttore in una funzione membro void privata o passare a `EXPECT_*()` se funziona. Questa sezione nella guida utente lo spiega.

12.14 La funzione SetUp() non viene chiamata. Perché?

Il C++ fa distinzione tra maiuscole e minuscole. È stato scritto `Setup()`?

Allo stesso modo, a volte le persone scrivono `SetUpTestSuite()` come `SetupTestSuite()` e si chiedono perché non venga mai chiamato.

12.15 Abbiamo diverse test suite che condividono la stessa logica della fixture; si deve definire una nuova classe fixture per ognuna di esse? Sembra piuttosto noioso.

Non è necessario. Invece di

```
class FooTest : public BaseTest {};

TEST_F(FooTest, Abc) { ... }
TEST_F(FooTest, Def) { ... }

class BarTest : public BaseTest {};

TEST_F(BarTest, Abc) { ... }
TEST_F(BarTest, Def) { ... }
```

si può semplicemente avere un `typedef` delle fixture:

```
typedef BaseTest FooTest;

TEST_F(FooTest, Abc) { ... }
TEST_F(FooTest, Def) { ... }

typedef BaseTest BarTest;

TEST_F(BarTest, Abc) { ... }
TEST_F(BarTest, Def) { ... }
```

12.16 Loutput di GoogleTest è sepolto in un sacco di messaggi di LOG. Cosa si deve fare?

Loutput di GoogleTest vuole essere un rapporto conciso e di facile comprensione. Se il test genera esso stesso un output testuale, si mescolerà con loutput di GoogleTest, rendendolo difficile da leggere. Tuttavia, esiste una soluzione semplice a questo problema.

Poiché i messaggi di LOG vanno a stderr, abbiamo deciso di lasciare che loutput di GoogleTest vada a stdout. In questo modo, si possono facilmente separare i due utilizzando il reindirizzamento. Per esempio:

```
$ ./my_test > gtest_output.txt
```

12.17 Perché dovrei preferire le fixture alle variabili globali?

Ci sono diversi buoni motivi:

1. È probabile che il test debba modificare gli stati delle sue variabili globali. Ciò rende difficile evitare che gli effetti collaterali sfuggano a un test e contaminino gli altri, rendendo difficile il debug. Utilizzando le fixture, ogni test ha un nuovo set di variabili diverse (ma con gli stessi nomi). Pertanto, i test vengono mantenuti indipendenti luno dallaltro.
2. Le variabili globali inquinano il namespace globale.
3. Le fixture possono essere riutilizzate tramite sottoclassi, cosa che non può essere eseguita facilmente con le variabili globali. Ciò è utile se molte test suite hanno qualcosa in comune.

12.18 Quale può essere largomento dellistruzione in ASSERT_DEATH()?

`ASSERT_DEATH(statement, matcher)` (o qualsiasi macro di asserzione death) può essere utilizzato ovunque sia valido *`statement`*. Quindi, in pratica *`statement`* può essere qualsiasi istruzione C++ che abbia senso nel contesto corrente. In particolare può fare riferimento a variabili globali e/o locali e può essere:

- una semplice chiamata di funzione (spesso accade),
- unespressione complessa, o
- unistruzione composta.

Alcuni esempi sono mostrati qui:

```
// A death test can be a simple function call.
TEST(MyDeathTest, FunctionCall) {
  ASSERT_DEATH(Xyz(5), "Xyz failed");
}

// Or a complex expression that references variables and functions.
TEST(MyDeathTest, ComplexExpression) {
  const bool c = Condition();
  ASSERT_DEATH((c ? Func1(0) : object2.Method("test")),
               "(Func1|Method) failed");
}

// Death assertions can be used anywhere in a function.  In
// particular, they can be inside a loop.
TEST(MyDeathTest, InsideLoop) {
  // Verifies that Foo(0), Foo(1), ..., and Foo(4) all die.
  for (int i = 0; i < 5; i++) {
    EXPECT_DEATH_M(Foo(i), "Foo has \\d+ errors",
                   ::testing::Message() << "where i is " << i);
  }
}

// A death assertion can contain a compound statement.
TEST(MyDeathTest, CompoundStatement) {
  // Verifies that at lease one of Bar(0), Bar(1), ..., and
```

(continues on next page)

(continua dalla pagina precedente)

```
  // Bar(4) dies.
  ASSERT_DEATH({
    for (int i = 0; i < 5; i++) {
      Bar(i);
    }
  },
  "Bar has \\d+ errors");
}
```

12.19 Ho una classe fixture FooTest, ma TEST_F(FooTest, Bar) mi dà lerrore "no matching function for call to `FooTest::FooTest()'". Perché?

GoogleTest deve essere in grado di creare oggetti della classe fixture, quindi deve avere un costruttore di default. Normalmente il compilatore ne definisce uno. Tuttavia, ci sono casi in cui è necessario definirne uno manualmente:

- Se si dichiara esplicitamente un costruttore non di default per la classe `FooTest` (`DISALLOW_EVIL_CONSTRUCTORS()` fa questo), allora si deve definire un costruttore di default, anche se vuoto.
- Se `FooTest` ha un membro dati const non statico, allora si deve definire il costruttore di default *e* inizializzare il membro const nellelenco degli inizializzatori del costruttore. (Le prime versioni di `gcc` non obbligano a inizializzare il membro const. È un bug corretto in `gcc 4`.)

12.20 Perché GoogleTest richiede che lintera test suite, anziché i singoli test, venga denominata *DeathTest quando usa ASSERT_DEATH?

GoogleTest non intercala test di diverse test suite. Cioè, esegue prima tutti i test in una test suite, poi esegue tutti i test nella test suite successiva e così via. GoogleTest fa questo perché deve impostare una test suite prima che venga eseguito il primo test al suo interno ed eseguirne il ńtear downż successivamente. La suddivisione del test case richiederebbe più processi di set-up e tear-down, il che è inefficiente e rende la semantica impura.

Se dovessimo determinare lordine dei test in base al nome del test invece che al nome del test case, avremmo un problema con la seguente situazione:

```
TEST_F(FooTest, AbcDeathTest) { ... }
TEST_F(FooTest, Uvw) { ... }

TEST_F(BarTest, DefDeathTest) { ... }
TEST_F(BarTest, Xyz) { ... }
```

Poiché `FooTest.AbcDeathTest` deve essere eseguito prima di `BarTest.Xyz`, e non interlacciamo test di diverse test suite, dobbiamo eseguire tutti i test nel caso `FooTest` prima di eseguire qualsiasi test nel caso `BarTest`. Ciò è in contraddizione con lobbligo di eseguire `BarTest.DefDeathTest` prima di `FooTest.Uvw`.

12.21 Ma non mi piace chiamare tutta la mia test suite *DeathTest quando contiene sia death test che non. Cosa si deve fare?

Non è necessario, ma volendo, si può suddividere la test suite in `FooTest` e `FooDeathTest`, dove i nomi chiariscono che sono correlati:

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

TEST_F(FooTest, Abc) { ... }
TEST_F(FooTest, Def) { ... }

using FooDeathTest = FooTest;

TEST_F(FooDeathTest, Uvw) { ... EXPECT_DEATH(...) ... }
TEST_F(FooDeathTest, Xyz) { ... ASSERT_DEATH(...) ... }
```

12.22 GoogleTest stampa i messaggi di LOG nel processo figlio di un death test solo quando il test fallisce. Come posso vedere i messaggi di LOG quando il death test ha successo?

Stampare i messaggi di LOG generati dallistruzione allinterno di `EXPECT_DEATH()` rende più difficile la ricerca di problemi reali nel log del genitore. Pertanto GoogleTest li stampa solo quando il death test è fallito.

Se cè davvero bisogno di vedere tali messaggi di LOG, una soluzione alternativa è quella di interrompere temporaneamente il death test (ad esempio modificando il pattern regex che dovrebbe corrispondere). Certo, questo è un trucco. Prenderemo in considerazione una soluzione più permanente dopo limplementazione dei death test in stile fork-and-exec.

12.23 Il compilatore dice `no match for 'operator«'` quando si usa unasserzione. Cosa dà?

Se si usa un tipo definito dallutente `FooType` in unasserzione, ci devessere una funzione `std::ostream& operator«(std::ostream&, const FooType&)` definita in modo tale da poter stampare un valore di `FooType`.

Inoltre, se `FooType` è dichiarato in un namespace, anche loperatore « deve essere definito nello *stesso* namespace. Vedere Tip of the Week #49 per i dettagli.

12.24 Come sopprimere i messaggi riguardo ai memory leak su Windows?

Poiché il singleton GoogleTest inizializzato staticamente richiede allocazioni nellheap, il rilevatore di memory leak di Visual C++ segnalerà le perdite di memoria alla fine dellesecuzione del programma. Il modo più semplice per evitare ciò è utilizzare le chiamate `_CrtMemCheckpoint` e `_CrtMemDumpAllObjectsSince` per non riportare alcun oggetto heap inizializzato staticamente. Consultare MSDN per ulteriori dettagli e routine aggiuntive di check/debug.

12.25 Come può il mio codice rilevare se è in esecuzione in un test?

Se si scrive codice che ńannusaż se è in esecuzione in un test e fa cose diverse di conseguenza, si sta perdendo la logica ńtest-onlyż nel codice di produzione e non esiste un modo semplice per garantire che i percorsi del codice test-only non vengano eseguiti per errore in produzione. Tale intelligenza porta anche agli Heisenbugs. Pertanto sconsigliamo vivamente questa pratica e GoogleTest non fornisce un modo per farlo.

In generale, il modo consigliato per far sì che il codice si comporti diversamente durante il test è la Dependency Injection. È possibile iniettare funzionalità diverse dal codice di test e quello di produzione. Poiché il codice di produzione non si collega affatto alla logica for-test (lattributo `testonly` per i target BUILD aiuta a garantirlo), non cè pericolo di eseguirlo accidentalmente.

Tuttavia, se *davvero*, *davvero*, *davvero* non si ha scelta e se si segue la regola di terminare i nomi dei programmi di test con `_test`, si può usare il trucco *orribile* di sniffare il nome delleseguibile (`argv[0]` in `main()`) per sapere se il codice è in fase di test.

12.26 Come disattivare temporaneamente un test?

Se si ha un test non funzionante che non si può correggere subito, si può aggiungere il prefisso `DISABLED_` al nome. Ciò lo escluderà dallesecuzione. Questo è meglio che commentare il codice o usare `#if 0`, poiché i test disabilitati vengono comunque compilati (e quindi non ńmarcirannoż).

Per includere i test disabilitati nellesecuzione del test, basta richiamare il programma di test con il flag `--gtest_also_run_disabled_tests`.

12.27 Va bene se ho due metodi di test `<TEST(Foo, Bar)` separati definiti in namespace diversi?

Sì.

La regola è che **tutti i metodi dei test nella stessa test suite devono utilizzare la stessa classe fixture**. Ciò significa che quanto segue è **consentito** poiché entrambi i test utilizzano la stessa classe fixture (`::testing::Test`).

```
namespace foo {
TEST(CoolTest, DoSomething) {
  SUCCEED();
}
}  // namespace foo

namespace bar {
TEST(CoolTest, DoSomething) {
  SUCCEED();
}
}  // namespace bar
```

Tuttavia, il codice seguente **non è consentito** e genererà un errore a runtime da GoogleTest poiché i metodi di test utilizzano classi di fixture diverse con lo stesso nome della test suite.

```
namespace foo {
class CoolTest : public ::testing::Test {};  // Fixture foo::CoolTest
TEST_F(CoolTest, DoSomething) {
  SUCCEED();
}
}  // namespace foo

namespace bar {
class CoolTest : public ::testing::Test {};  // Fixture: bar::CoolTest
TEST_F(CoolTest, DoSomething) {
  SUCCEED();
}
}  // namespace bar
```

CAPITOLO 13

Domande frequenti su gMock Legacy

13.1 Quando chiamo un metodo sul mio oggetto mock, viene invece richiamato il metodo per loggetto reale. Qual è il problema?

Affinché un metodo possa essere mock-ato, deve essere *virtual*, a meno che non si utilizzi la high-perf dependency injection technique.

13.2 Posso mock-are una funzione variadica?

Non è possibile mock -are una funzione variadica (ovvero una funzione che accetta argomenti con puntini di sospensione `...`) direttamente in gMock.

Il problema è che, in generale, non cè *nessun modo* per un oggetto mock di sapere quanti argomenti vengono passati al metodo variadico e quali sono i tipi degli argomenti. Solo l*autore della classe base* conosce il protocollo e non possiamo guardargli nella testa.

Pertanto, per simulare una funzione del genere, l*utente* deve insegnare alloggetto mock-ato come calcolare il numero di argomenti e i loro tipi. Un modo per farlo è fornire versioni sovraccaricate [overloaded] della funzione.

Gli argomenti con i puntini di sospensione sono ereditati dal C e non sono realmente una funzionalità del C++. Non sono sicuri da usare e non funzionano con argomenti che hanno costruttori o distruttori. Pertanto consigliamo di evitarli il più possibile in C++.

13.3 MSVC mi dà un warning C4301 o C4373 quando definisco un metodo mock con un parametro const. Perché?

Se lo compili utilizzando Microsoft Visual C++ 2005 SP1:

```
class Foo {
  ...
  virtual void Bar(const int i) = 0;
};

class MockFoo : public Foo {
```

(continues on next page)

(continua dalla pagina precedente)

```
  ...
  MOCK_METHOD(void, Bar, (const int i), (override));
};
```

Si potrebbe ricevere il seguente warning:

```
warning C4301: 'MockFoo::Bar': overriding virtual function only differs from 'Foo::Bar
↪' by const/volatile qualifier
```

Questo è un bug di MSVC. Lo stesso codice viene compilato bene con gcc, ad esempio. Se si usa Visual C++ 2008 SP1, si riceverà il warning:

```
warning C4373: 'MockFoo::Bar': virtual function overrides 'Foo::Bar', previous␣
↪versions of the compiler did not override when parameters only differed by const/
↪volatile qualifiers
```

In C++, se si *dichiara* una funzione con un parametro `const`, il modificatore `const` viene ignorato. Pertanto, la classe base `Foo` sopra è equivalente a:

```
class Foo {
  ...
  virtual void Bar(int i) = 0;  // int or const int?  Makes no difference.
};
```

Infatti, si può *dichiarare* `Bar()` con un parametro `int` e definirlo con un parametro `const int`. Il compilatore li abbinerà comunque.

Poiché la creazione di un parametro `const` non ha significato nella dichiarazione del metodo, consigliamo di rimuoverlo sia in `Foo` che in `MockFoo`. Ciò dovrebbe risolvere il bug VC.

Si noti che qui stiamo parlando del modificatore *top-level* `const`. Se il parametro della funzione viene passato tramite puntatore o riferimento, dichiarare il puntato o il riferito come `const` ha ancora senso. Ad esempio, le due dichiarazioni seguenti *non* sono equivalenti:

```
void Bar(int* p);         // Neither p nor *p is const.
void Bar(const int* p);  // p is not const, but *p is.
```

13.4 Non riesco a capire perché gMock pensi che le mie aspettative non siano soddisfatte. Cosa dovrei fare?

Potresti voler eseguire il test con `--gmock_verbose=info`. Questo flag consente a gMock di stampare un trace per ogni chiamata di funzione mock che riceve. Studiando il trace, si otterranno informazioni sul motivo per cui le aspettative [expectation] impostate non vengono soddisfatte.

Se si vede il messaggio ńThe mock function has no default action set, and its return type has no default value set.ż, provare ad aggiungere unazione di default. A causa di un problema noto, le chiamate impreviste ai mock senza azioni di default non stampano un confronto dettagliato tra gli argomenti effettivi e quelli previsti.

13.5 Il mio programma è crashato e `ScopedMockLog` ha inviato tonnellate di messaggi. È un bug di gMock?

gMock e `ScopedMockLog` probabilmente stanno facendo la cosa giusta in questo caso.

Quando un test va in crash, lhandler del segnale di errore tenterà di loggare molte informazioni (il trace dello stack e la mappa degli indirizzi, ad esempio). I messaggi sono complessi se sono presenti molti thread con stack profondi. Quando `ScopedMockLog` intercetta questi messaggi e rileva che non corrispondono ad alcuna aspettativa, stampa un errore per ciascuno di essi.

Si può imparare a ignorare gli errori oppure si possono riscrivere le aspettative per rendere il test più robusto, ad esempio aggiungendo qualcosa come:

```
using ::testing::AnyNumber;
using ::testing::Not;
...
  // Ignores any log not done by us.
  EXPECT_CALL(log, Log(_, Not(EndsWith("/my_file.cc")), _))
      .Times(AnyNumber());
```

13.6 Come posso asserire che una funzione non viene MAI chiamata?

```
using ::testing::_;
...
  EXPECT_CALL(foo, Bar(_))
      .Times(0);
```

13.7 Ho un test fallito in cui gMock mi dice DUE VOLTE che una particolare expectation non è soddisfatta. Non è ridondante?

Quando gMock rileva un errore, stampa le informazioni rilevanti (gli argomenti della funzione mock, lo stato delle expectation rilevanti, ecc.) per aiutare lutente a eseguire il debug. Se viene rilevato un altro errore, gMock farà lo stesso, inclusa la stampa dello stato delle expectation rilevanti.

A volte lo stato di una expectation non cambia tra due fallimenti e si vedrà la stessa descrizione dello stato due volte. Tuttavia *non* sono ridondanti, poiché si riferiscono a *diversi momenti nel tempo*. Il fatto che siano la stessa cosa *è* uninformazione interessante.

13.8 Ho un errore sul check dellheap quando utilizzo un oggetto mock, ma usando un oggetto reale va bene. Cosa può esserci di sbagliato?

La classe (si spera uninterfaccia pura) che si sta mock-ando ha un distruttore virtuale?

Ogni volta che si deriva da una classe base, assicurarsi che il suo distruttore sia virtuale. Altrimenti Accadranno Cose Brutte. Si consideri il seguente codice:

```
class Base {
 public:
  // Not virtual, but should be.
  ~Base() { ... }
  ...
};

class Derived : public Base {
 public:
  ...
 private:
  std::string value_;
};

...
  Base* p = new Derived;
```

(continues on next page)

(continua dalla pagina precedente)

```
  ...
  delete p;  // Surprise! ~Base() will be called, but ~Derived() will not
                // - value_ is leaked.
```

Cambiando `~Base()` in virtual, `~Derived()` verrà chiamato correttamente quando viene eseguito `delete p` e il checker dellheap sarà soddisfatto.

13.9 La regola ńLe expectation più nuove sovrascrivono quelle più vecchież complica la scrittura delle expectation. Perché gMock lo fa?

Quando le persone si lamentano di questo, spesso si riferiscono a codici come:

```
using ::testing::Return;
...
  // foo.Bar() should be called twice, return 1 the first time, and return
  // 2 the second time.  However, I have to write the expectations in the
  // reverse order.  This sucks big time!!!
  EXPECT_CALL(foo, Bar())
      .WillOnce(Return(2))
      .RetiresOnSaturation();
  EXPECT_CALL(foo, Bar())
      .WillOnce(Return(1))
      .RetiresOnSaturation();
```

Il problema è che non hanno scelto il modo **migliore** per esprimere lintento del test.

Per default, non è necessario che le expectation corrispondano in *ogni* particolare ordine. Se si vuole che corrispondano in un certo ordine, si deve essere espliciti. Questa è la filosofia fondamentale di gMock (e di jMock): è facile sovra-specificare accidentalmente i test e noi vogliamo rendere più difficile farlo.

Esistono due modi migliori per scrivere le specifiche del test. Si potrebbero mettere le expectation in sequenza:

```
using ::testing::Return;
...
  // foo.Bar() should be called twice, return 1 the first time, and return
  // 2 the second time.  Using a sequence, we can write the expectations
  // in their natural order.
  {
    InSequence s;
    EXPECT_CALL(foo, Bar())
        .WillOnce(Return(1))
        .RetiresOnSaturation();
    EXPECT_CALL(foo, Bar())
        .WillOnce(Return(2))
        .RetiresOnSaturation();
  }
```

oppure si può mettere la sequenza di azioni nella stessa expectation:

```
using ::testing::Return;
...
  // foo.Bar() should be called twice, return 1 the first time, and return
  // 2 the second time.
  EXPECT_CALL(foo, Bar())
      .WillOnce(Return(1))
```

(continues on next page)

(continua dalla pagina precedente)

```
    .WillOnce(Return(2))
    .RetiresOnSaturation();
```

Torniamo alle domande originali: perché gMock cerca le expectation (e le `ON_CALL`) da dietro in avanti? Perché ciò consente allutente di impostare in anticipo il comportamento di un mock per il caso comune (ad esempio nel costruttore del mock o nella fase di impostazione della fixture) e di personalizzarlo con regole più specifiche in seguito. Se gMock effettua la ricerca dalla parte anteriore a quella posteriore, questo modello molto utile non sarà possibile.

13.10 gMock stampa un warning quando viene chiamata una funzione senza EXPECT_CALL, anche se ho impostato il suo comportamento utilizzando ON_CALL. Sarebbe ragionevole non mostrare il warning in questo caso?

Quando scegliamo tra lessere puliti e lessere al sicuro, tendiamo verso questultimo. Quindi la risposta è che pensiamo sia meglio mostrare il warning.

Spesso le persone scrivono `ON_CALL` nel costruttore delloggetto mock o in `SetUp()`, poiché il comportamento di default raramente cambia da test a test. Poi nel corpo del test si stabiliscono le expectation, che spesso sono diverse per ogni test. Avere un `ON_CALL` nella parte del set-up di un test non significa che le chiamate siano previste [expected]. Se non cè `EXPECT_CALL` e il metodo viene chiamato, probabilmente si tratta di un errore. Se lasciamo passare la chiamata senza notificare allutente, i bug potrebbero insinuarsi inosservati.

Se invece si è sicuri che le chiamate siano andate bene si può scrivere

```
using ::testing::_;
...
  EXPECT_CALL(foo, Bar(_))
      .WillRepeatedly(...);
```

invece di

```
using ::testing::_;
...
  ON_CALL(foo, Bar(_))
      .WillByDefault(...);
```

Questo dice a gMock che ci si aspettano le chiamate e non dovrebbe essere stampato alcun warning.

Inoltre, si può controllare la verbosità specificando `--gmock_verbose=error`. Altri valori sono `info` e `warning`. Se si ritiene che loutput sia troppo prolisso durante il debug, si sceglie semplicemente un livello meno dettagliato di verbosità.

13.11 Come si può eliminare largomento della funzione mock in unazione?

Se la funzione mock accetta un argomento puntatore e si vuole eliminare quellargomento, si può utilizzare testing::DeleteArg() per il delete dellN-esimo argomento (partendo da zero):

```
using ::testing::_;
  ...
  MOCK_METHOD(void, Bar, (X* x, const Y& y));
  ...
  EXPECT_CALL(mock_foo_, Bar(_, _))
      .WillOnce(testing::DeleteArg<0>()));
```

13.12 Come posso eseguire unazione arbitraria sullargomento di una funzione mock?

Se si deve eseguire qualche azione non supportata direttamente da gMock, ci si ricordi che si possono definire le proprie azioni utilizzando *MakeAction()* o *MakePolymorphicAction()*, oppure si può scrivere una funzione stub e invocarla utilizzando *Invoke()*.

```
using ::testing::_;
using ::testing::Invoke;
  ...
  MOCK_METHOD(void, Bar, (X* p));
  ...
  EXPECT_CALL(mock_foo_, Bar(_))
      .WillOnce(Invoke(MyAction(...)));
```

13.13 Il mio codice chiama una funzione statica/globale. Posso renderla mock?

Si può, ma si devono apportare alcune modifiche.

In generale, se si deve rendere mock una funzione statica, è segno che i moduli sono accoppiati troppo strettamente (e meno flessibili, meno riutilizzabili, meno testabili, ecc.). Probabilmente sarebbe meglio definire una piccola interfaccia e chiamare la funzione attraverso quellinterfaccia, che quindi può essere facilmente resa mock. Inizialmente Cè un po di lavoro, ma di solito si ripaga rapidamente.

Questo post del Google Testing Blog lo dice in modo eccellente. Consultarlo.

13.14 Il mio oggetto mock deve fare cose complesse. È molto faticoso specificare le azioni. gMock fa schifo!

So che non è una domanda, ma riceverai comunque una risposta gratuitamente. :-)

Con gMock puoi creare facilmente mock in C++. E le persone potrebbero essere tentate di usarli ovunque. A volte funzionano alla grande, a volte potresti trovarli, beh, difficili da usare. Quindi, cosa cè che non va in questultimo caso?

Quando si scrive un test senza utilizzare mock, si sollecita il codice e si afferma che restituisce il valore corretto o che il sistema è in uno stato previsto. Questo è talvolta chiamato ńtest basato sullo statoż.

I mock sono ottimi per quello che alcuni chiamano test ńbasati sullinterazioneż: invece di controllare lo stato del sistema alla fine, gli oggetti mock verificano che siano invocati nel modo giusto e segnalano un errore non appena si verifica, dando un controllo preciso sul contesto in cui si è verificato lerrore. Questo è spesso più efficace ed economico rispetto ai test state-based.

Se si eseguono test basati sullo stato e si utilizza un test doppione solo per simulare loggetto reale, probabilmente è meglio utilizzare un falso. Usare un mock in questo caso è faticoso, poiché non è un punto di forza per i mock eseguire azioni complesse. Se ci si trova in questo caso e si pensa che i mock facciano schifo, semplicemente non si sta usando lo strumento giusto per il problema. Oppure si sta cercando di risolvere il problema sbagliato. :-)

13.15 Ho ricevuto un warning ńUninteresting function call encountered - default action taken..ż Devo andare nel panico?

Certamente NO! E solo per tua informazione. :-)

Ciò che si intende è che si ha una funzione mock, non è stata impostata alcuna expectation su di essa (secondo la regola di gMock ciò significa che non si è interessati alle chiamate a questa funzione e quindi può essere chiamata un numero qualsiasi di volte), e viene chiamata. Va bene: non è stato detto che non è OK chiamare la funzione!

Cosa succederebbe se in realtà si intendesse impedire la chiamata di questa funzione, ma ci si dimenticasse di scrivere `EXPECT_CALL(foo, Bar()).Times(0)`? Anche se si può sostenere che è colpa dellutente, gMock cerca di essere gentile e stampa una nota.

Quindi, quando si vede il messaggio e si crede che non dovrebbero esserci chiamate poco interessanti, si dovrebbe indagare su cosa sta succedendo. Per semplificarci la vita, gMock scarica lanalisi dello stack quando viene incontrata una chiamata poco interessante. Da ciò si può capire quale funzione mock è e come viene chiamata.

13.16 Voglio definire unazione personalizzata. Dovrei usare Invoke() o implementare linterfaccia ActionInterface?

Va bene in ogni caso: si scelga quello più conveniente per le proprie circostanze.

Di solito, se lazione riguarda un particolare tipo di funzione, definirla utilizzando `Invoke()` dovrebbe essere più semplice; se lazione può essere utilizzata in funzioni di diverso tipo (ad esempio si sta definendo `Return(*value*)`), `MakePolymorphicAction()` è più semplice. A volte si vuole un controllo preciso sui tipi delle funzioni in cui può essere utilizzata lazione e implementare `ActionInterface` è la strada da percorrere in questo caso. Vedere limplementazione di `Return()` in `gmock-actions.h` per un esempio.

13.17 Io uso SetArgPointee() in WillOnce(), ma gcc si lamenta di ńconflicting return type specifiedż. Cosa significa?

Hai ricevuto questo errore poiché gMock non ha idea di quale valore dovrebbe restituire quando viene chiamato il metodo mock. `SetArgPointee()` dice qual è leffetto collaterale, ma non dice quale dovrebbe essere il valore restituito. È necessario che `DoAll()` concateni un `SetArgPointee()` con un `Return()` che fornisca un valore appropriato allAPI mock-ata.

Vedere questa ricetta per maggiori dettagli e un esempio.

13.18 Ho unenorme classe mock e Microsoft Visual C++ va in ńout of memoryż durante la compilazione. Cosa posso fare?

Abbiamo notato che quando viene utilizzato il flag del compilatore `/clr`, Visual C++ usa 5~6 volte più memoria durante la compilazione di una classe mock. Suggeriamo di evitare `/clr` durante la compilazione di mock C++ nativi.

CAPITOLO 14

Esempi di Googletest

Sarete certamente interessati a dare unocchiata agli esempi di googletest. La directory ńsamplesż contiene una serie di esempi ben commentati che mostrano come utilizzare diverse funzioni di googletest.

- Sample #1 mostra i passaggi fondamentali per utilizzare googletest per testare le funzioni C++.
- Sample #2 mostra una unit test più complessa per una classe con più funzioni membro.
- Sample #3 usa una fixture.
- Sample #4 insegna come utilizzare googletest e `googletest.h` insieme per ottenere il meglio da entrambe le librerie.
- Sample #5 inserisce la logica di test condivisi in una fixture di base e la riutilizza in fixture derivate.
- Sample #6 illustra i test con tipi parametrizzati.
- Sample #7 insegna le basi dei test valori parametrizzati.
- Sample #8 mostra lutilizzo di `Combine()` in test con valori parametrizzati.
- Sample #9 mostra lutilizzo dellAPI listener per modificare loutput della console di Google Test e lutilizzo della relativa API di [reflection] (riflessione) per esaminare i risultati del test.
- Sample #10 mostra luso dellAPI del listener per implementare un controllo primitivo per i memory leak.

CAPITOLO 15

Utilizzo di GoogleTest da vari sistemi di build

GoogleTest viene fornito con file pkg-config utilizzabili per determinare tutti i flag necessari per la compilazione e il link a GoogleTest (e a GoogleMock). Pkg-config è un semplice formato di testo standardizzato contenente

- Il path di includedir (-I)
- le definizioni di macro necessarie (-D)
- ulteriori flag richiesti (-pthread)
- il path della libreria (-L)
- la libreria da linkare (-l)

Tutti i sistemi di build attuali supportano pkg-config in un modo o nellaltro. Per tutti gli esempi qui assumiamo che si voglia compilare lesempio `samples/sample3_unittest.cc`.

15.1 CMake

Luso di `pkg-config` in CMake è abbastanza semplice:

```
find_package(PkgConfig)
pkg_search_module(GTEST REQUIRED gtest_main)

add_executable(testapp)
target_sources(testapp PRIVATE samples/sample3_unittest.cc)
target_link_libraries(testapp PRIVATE ${GTEST_LDFLAGS})
target_compile_options(testapp PRIVATE ${GTEST_CFLAGS})

enable_testing()
add_test(first_and_only_test testapp)
```

In genere è consigliabile utilizzare `target_compile_options` + `_CFLAGS` su `target_include_directories` + `_INCLUDE_DIRS` poiché il primo include non solo i flag -I (GoogleTest potrebbe richiedere una macro che indichi agli header interni che tutte le librerie sono state compilate con il threading abilitato. Inoltre, GoogleTest potrebbe anche richiedere `-pthread` nella fase di compilazione e, di conseguenza, dividere la variabile `Cflags` di pkg-config in include dir e macro per `target_compile_definitions()` potrebbe ancora perderlo). La stessa raccomandazione vale per lutilizzo di `_LDFLAGS` al posto del più comune `_LIBRARIES`, che scarta i flag `-L` e `-pthread`.

15.2 Aiuto! pkg-config non trova GoogleTest!

Supponiamo che si abbia un `CMakeLists.txt` sulla falsariga di quello in questo tutorial e si provi a eseguire `cmake`. È molto probabile che si verifichi un problema sulla falsariga di:

```
-- Checking for one of the modules 'gtest_main'
CMake Error at /usr/share/cmake/Modules/FindPkgConfig.cmake:640 (message):
  None of the required 'gtest_main' found
```

Questi errori sono comuni se GoogleTest è stato installato da solo e non lo fa parte di una distribuzione o di un altro gestore di pacchetti. Se è così, si deve dire a pkg-config dove può trovare i file `.pc` contenenti le informazioni. Supponiamo che GoogleTest sia installato su `/usr/local`, è possibile che i file `.pc` siano installati in `/usr/local/lib64/pkgconfig`. Se si imposta

```
export PKG_CONFIG_PATH=/usr/local/lib64/pkgconfig
```

pkg-config proverà anche a cercare in `PKG_CONFIG_PATH` per trovare `gtest_main.pc`.

15.3 Uso di pkg-config in una impostazione di cross-compilazione

Pkg-config è utilizzabile anche in un ambiente di cross-compilazione. Per farlo, supponiamo che il prefisso finale dellinstallazione cross-compilata sia `/usr` e che il sysroot sia `/home/MYUSER/sysroot`. Configurare e installare GTest con

```
mkdir build && cmake -DCMAKE_INSTALL_PREFIX=/usr ..
```

Installare in sysroot con `DESTDIR`:

```
make -j install DESTDIR=/home/MYUSER/sysroot
```

Prima di continuare, si consiglia di definire **sempre** le seguenti due variabili per pkg-config in unimpostazione di cross-compilazione:

```
export PKG_CONFIG_ALLOW_SYSTEM_CFLAGS=yes
export PKG_CONFIG_ALLOW_SYSTEM_LIBS=yes
```

altrimenti `pkg-config` filtrerà i flag `-I` e `-L` rispetto ai prefissi standard come `/usr` (vedere https://bugs.freedesktop.org/show_bug.cgi?id=28264#c3 per i motivi per cui di solito è necessario eseguire questa rimozione).

Guardando il file pkg-config generato, sarà simile a

```
libdir=/usr/lib64
includedir=/usr/include

Name: gtest
Description: GoogleTest (without main() function)
Version: 1.11.0
URL: https://github.com/google/googletest
Libs: -L${libdir} -lgtest -lpthread
Cflags: -I${includedir} -DGTEST_HAS_PTHREAD=1 -lpthread
```

Notare che sysroot non è incluso in `libdir` e in `includedir`! Provando ad eseguire `pkg-config` col `PKG_CONFIG_LIBDIR=/home/MYUSER/sysroot/usr/lib64/pkgconfig` corretto rispetto a questo file `.pc`, si otterrà

```
$ pkg-config --cflags gtest
-DGTEST_HAS_PTHREAD=1 -lpthread -I/usr/include
$ pkg-config --libs gtest
-L/usr/lib64 -lgtest -lpthread
```

che è ovviamente sbagliato e punta alla radice di `CBUILD` e non di `CHOST`. Per usarlo in unimpostazione di cross-compilazione, dobbiamo dire a pkg-config di inserire lattuale sysroot nelle variabili `-I` e `-L`. Diciamo ora a pkg-config il vero sysroot

```
export PKG_CONFIG_DIR=
export PKG_CONFIG_SYSROOT_DIR=/home/MYUSER/sysroot
export PKG_CONFIG_LIBDIR=${PKG_CONFIG_SYSROOT_DIR}/usr/lib64/pkgconfig
```

ed eseguendo nuovamente `pkg-config` otteniamo

```
$ pkg-config --cflags gtest
-DGTEST_HAS_PTHREAD=1 -lpthread -I/home/MYUSER/sysroot/usr/include
$ pkg-config --libs gtest
-L/home/MYUSER/sysroot/usr/lib64 -lgtest -lpthread
```

che ora contiene il sysroot corretto. Per una guida più completa su come includere anche `${CHOST}` nelle chiamate di sistema della build, vedere leccellente tutorial di Diego Elio Pettenò: https://autotools.io/pkgconfig/cross-compiling.html

CAPITOLO 16

Documentazione Creata dalla Comunità

Di seguito è riportato un elenco, senza un ordine particolare, dei link alla documentazione creata dalla community di Googletest.

- Googlemock Insights, di ElectricRCAircraftGuy