www.italiancpp.org

Sviluppo di un framework di unit-test C++

Gianfranco Zuliani

Bologna, 8 Novembre 2014

Origini

```
#include ...


int main() {

    // test body









  return 0;
}
```

Test file

```
#include ...


int main() {

    // test body
    // exit with error if:
    //  * an exception is thrown
    //  * an assertion fails




  return 0;
}
```

Test file

```
#include ...


int main() {
  try {
    // test body
    // exit with error if:
    //  * an exception is thrown
    //  * an assertion fails
  } catch (...) {
    std::cerr << "test failed" << std::endl;
    return 1;
  }
  std::cerr << "test succeeded" << std::endl;
  return 0;
}
```

Test file

```
#include ...


int main() {
  try {
    // test body
    // exit with error if:
    //  * an exception is thrown [OK]
    //  * an assertion fails
  } catch (...) {
    std::cerr << "test failed" << std::endl;
    return 1;
  }
  std::cerr << "test succeeded" << std::endl;
  return 0;
}
```

Test file

```
#include ...


int main() {
  try {
    // test body
    // exit with error if:
    //  * an exception is thrown [OK]
    //  * an assertion fails     [OK] if an assertion raises an exception!
  } catch (...) {
    std::cerr << "test failed" << std::endl;
    return 1;
  }
  std::cerr << "test succeeded" << std::endl;
  return 0;
}
```

Test file

```
#include ...


int main() {
  try {
    // test body
    // exit with error if:
    //  * an exception is thrown [OK]
    //  * an assertion fails     [OK]
  } catch (...) {
    std::cerr << "test failed" << std::endl;
    return 1;
  }
  std::cerr << "test succeeded" << std::endl;
  return 0;
}
```

Test file

```
#include ...


int main() {
  try {
    // test body



  } catch (...) {
    std::cerr << "test failed" << std::endl;
    return 1;
  }
  std::cerr << "test succeeded" << std::endl;
  return 0;
}
```

Test framework [test.h]

```
#include <iostream>

#define TEST_BEGIN \
int main() { \
  try {



#define TEST_END \
  } catch (...) { \
    std::cerr << "test failed" << std::endl; \
    return 1; \
  } \
  std::cerr << "test succeeded" << std::endl; \
  return 0; \
}
```

Test file

```
#include ...


int main() {
  try {
    // test body



  } catch (...) {
    std::cerr << "test failed" << std::endl;
    return 1;
  }
  std::cerr << "test succeeded" << std::endl;
  return 0;
}
```

Test file

```
#include ...
#include <test.h>

int main() {
  try {
    // test body



  } catch (...) {
    std::cerr << "test failed" << std::endl;
    return 1;
  }
  std::cerr << "test succeeded" << std::endl;
  return 0;
}
```

Test file

```
#include ...
#include <test.h>

TEST_BEGIN

    // test body



  } catch (...) {
    std::cerr << "test failed" << std::endl;
    return 1;
  }
  std::cerr << "test succeeded" << std::endl;
  return 0;
}
```

Test file

```
#include ...
#include <test.h>

TEST_BEGIN

    // test body



TEST_END
```

ASSERT

```
#include ...
#include <test.h>

TEST_BEGIN

    // ...
    ASSERT(expr);


TEST_END
```

ASSERT

```
#include ...
#include <test.h>

TEST_BEGIN

    // ...
    ASSERT(i == j);


TEST_END
```

Test framework [test.h]

```
#include <iostream>
#include <stdexcept>

// ...

struct test_error : public std::logic_error {
  test_error() : std::logic_error("test error") { }
};
```

Test framework [test.h]

```
#include <iostream>
#include <stdexcept>

// ...

struct test_error : public std::logic_error {
  test_error() : std::logic_error("test error") { }
};

#define ASSERT(expr_) \
do { \
  if (!(expr_)) { \
    std::cerr << __FILE__ << "(" << __LINE__ << ") : test error - " << #expr_ << std::endl; \
    throw test_error(); \
  } \
} while (0)
```

Test framework [test.h]

```
#include <iostream>
#include <stdexcept>

// ...

struct test_error : public std::logic_error {
  test_error() : std::logic_error("test error") { }
};

#define ASSERT(expr_) \
do { \
  if (!(expr_)) { \
    std::cerr << __FILE__ << "(" << __LINE__ << ") : test error - " << #expr_ << std::endl; \
    throw test_error(); \
  } \
} while (0)
```

ASSERT

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  ASSERT(i == j);


TEST_END
```

ASSERT

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  ASSERT(i == j);


TEST_END
```

```
d:\projects\...\test_file(42) : test error - i == j
test failed
```

Pattern “TestThrow”

```
#include ...
#include <test.h>

TEST_BEGIN

    // code that should throw



TEST_END
```

Pattern “TestThrow”

```
#include ...
#include <test.h>

TEST_BEGIN
  try {
    // code that should throw
    ASSERT(false);
  } catch(const expected_exception&) {
  }
TEST_END
```

Pattern “TestThrow”

```
#include ...
#include <test.h>

TEST_BEGIN
  try {
    // code that should throw
    ASSERT(false);
  } catch(const expected_exception&) {
  }
TEST_END
```

```
d:\projects\...\test_file.cpp(42) : test error - false
test failed
```

Fondamenti

```
#include <test.h>

TEST_BEGIN

  ASSERT(...);

TEST_END
```

Limiti

- Messaggi d'errore poco circostanziati

Limiti

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  ASSERT(i == j);


TEST_END
```

Limiti

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  ASSERT(i == j);


TEST_END
```

```
d:\projects\...\test_file.cpp(42) : test error - i == j
test failed
```

Limiti

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  ASSERT(i == j);


TEST_END
```

```
d:\projects\...\test_file.cpp(42) : test error - i == j <--- i=?, j=?
test failed
```

Limiti

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  std::cerr << i << ", " << j << std::endl;
  ASSERT(i == j);

TEST_END
```

Limiti

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  std::cerr << i << ", " << j << std::endl;
  ASSERT(i == j);

TEST_END
```

```
1, 2
d:\projects\...\test_file.cpp(42) : test error - i == j
test failed
```

Limiti

- Messaggi d'errore poco circostanziati

Limiti

- Messaggi d'errore poco circostanziati
- Pattern “TestThrow” troppo prolisso

Limiti

```
#include ...
#include <test.h>

TEST_BEGIN
  try {
    // code that should throw
    ASSERT(false);
  } catch(const test_exception&) {
  }
TEST_END
```

Limiti

```
#include ...
#include <test.h>

TEST_BEGIN
  try {
    // code that should throw
    ASSERT(false);
  } catch(const test_exception&) {
  }
TEST_END
```

Limiti

```
#include ...
#include <test.h>

TEST_BEGIN

    THROWS(code_that_should_throw());



TEST_END
```

Limiti

- Messaggi d'errore poco circostanziati
- Pattern “TestThrow” piuttosto prolisso

Limiti

- Messaggi d'errore poco circostanziati
- Pattern “TestThrow” piuttosto prolisso
- Uscita anticipata dal test in caso d'errore

Limiti

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  ASSERT(i == 0);
  ASSERT(j == 0);

TEST_END
```

Limiti

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  ASSERT(i == 0); // fails...
  ASSERT(j == 0);

TEST_END
```

```
d:\projects\...\test_file.cpp(42) : test error - i == 0
test failed
```

Limiti

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  ASSERT(i == 0); // fails...
  ASSERT(j == 0); // not evaluated!

TEST_END
```

```
d:\projects\...\test_file.cpp(42) : test error - i == 0
test failed
```

Limiti

- Messaggi d'errore poco circostanziati
- Pattern “TestThrow” piuttosto prolisso
- Uscita anticipata dal test in caso d'errore

Limiti

- Messaggi d'errore poco circostanziati
- Pattern “TestThrow” piuttosto prolisso
- Uscita anticipata dal test in caso d'errore
- Test monolitico

Test monolitico

```
#include <gut.h>
#include ...

TEST_BEGIN

  // test body



TEST_END
```

Test monolitico

```
#include <gut.h>
#include <string-stack.h>

TEST_BEGIN

  // test body



TEST_END
```

Test monolitico

```
#include <gut.h>
#include <string-stack.h>

TEST_BEGIN

  StringStack stack = StringStack();
  ASSERT(stack.empty());

TEST_END
```

Test monolitico

```
#include <gut.h>
#include <string-stack.h>

TEST_BEGIN

  StringStack stack = StringStack();
  ASSERT(stack.empty());

  std::string aString = "Only String";
  stack.push(aString);
  ASSERT(!stack.empty());

TEST_END
```

Test monolitico

```
#include <gut.h>
#include <string-stack.h>

TEST_BEGIN

  StringStack stack = StringStack();
  ASSERT(stack.empty());

  std::string aString = "Only String";
  stack.push(aString);
  ASSERT(!stack.empty());

  std::string topValue = stack.top();
  ASSERT(aString == topValue);
  ASSERT(!stack.empty());



TEST_END
```

Test monolitico

```
#include <gut.h>
#include <string-stack.h>

TEST_BEGIN

  StringStack stack = StringStack();
  ASSERT(stack.empty());

  std::string aString = "Only String";
  stack.push(aString);
  ASSERT(!stack.empty());

  std::string topValue = stack.top();
  ASSERT(aString == topValue);
  ASSERT(!stack.empty());

  stack.pop();
  ASSERT(stack.empty());

TEST_END
```

Test monolitico

```
#include <gut.h>
#include <string-stack.h>

TEST_BEGIN

  StringStack stack = StringStack();
  ASSERT(stack.empty());

  std::string aString = "Only String";
  stack.push(aString);
  ASSERT(!stack.empty());

  std::string topValue = stack.top();
  ASSERT(aString == topValue);
  ASSERT(!stack.empty());

  stack.pop();
  ASSERT(stack.empty());

TEST_END
```

Limiti

- Messaggi d'errore poco circostanziati
- Pattern “TestThrow” piuttosto prolisso
- Uscita anticipata dal test in caso d'errore
- Test monolitico

Limiti

- Messaggi d'errore poco circostanziati
- Pattern “TestThrow” piuttosto prolisso
- Uscita anticipata dal test in caso d'errore
- Test monolitico
- Prospetto finale cablato

Limiti

- Messaggi d'errore poco circostanziati
- Pattern “TestThrow” piuttosto prolisso
- Uscita anticipata dal test in caso d'errore
- Test monolitico
- Prospetto finale cablato

ASSERT /2

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  ASSERT(i == j);


TEST_END
```

ASSERT /2

```
#include ...
#include <test.h>

TEST_BEGIN

  // ...
  ASSERT(i == j);


TEST_END
```

d:\projects\...\test_file.cpp(42) : test error - i == j
test failed

ASSERT /2

```
#include ...
#include <gut.h>

TEST_BEGIN

  // ...
  ASSERT(i == j);


TEST_END
```

```
d:\projects\...\test_file.cpp(42) : test error - i == j evaluates to 1 == 2
test failed
```

ASSERT /2

```
#define ASSERT(expr_) \
do { \
  if (!(expr_)) { \
    std::cerr << ... << #expr_ << std::endl; \
    throw test_error(); \
  } \
} while (0)
```

ASSERT /2

```
#define ASSERT(expr_) \
do { \
  if (!(Capture()->*expr_)) { \
    std::cerr << ... << #expr_ << " evaluates to " << last_expr_ << std::endl; \
    throw test_error(); \
  } \
} while (0)
```

ASSERT /2

```
#define ASSERT(expr_) \
do { \
  if (!(Capture()->*expr_)) { \
    std::cerr << ... << #expr_ << " evaluates to " << last_expr_ << std::endl; \
    throw test_error(); \
  } \
} while (0)

ASSERT(i == j)
```

ASSERT /2

```
#define ASSERT(expr_) \
do { \
  if (!(Capture()->*expr_)) { \
    std::cerr << ... << #expr_ << " evaluates to " << last_expr_ << std::endl; \
    throw test_error(); \
  } \
} while (0)

ASSERT(i == j)
do { \
  if (!(Capture()->*expr_)) { \
    std::cerr << ... << #expr_   << " evaluates to " << last_expr_ << std::endl; \
    throw test_error(); \
  } \
} while (0)
```

ASSERT /2

```
#define ASSERT(expr_) \
do { \
  if (!(Capture()->*expr_)) { \
    std::cerr << ... << #expr_ << " evaluates to " << last_expr_ << std::endl; \
    throw test_error(); \
  } \
} while (0)
```

```
ASSERT(i == j)
do { \
  if (!(Capture()->*i == j)) { \
    std::cerr << ... << "i == j" << " evaluates to " << last_expr_ << std::endl; \
    throw test_error(); \
  } \
} while (0)
```

ASSERT /2

```
#define ASSERT(expr_) \
do { \
  if (!(Capture()->*expr_)) { \
    std::cerr << ... << #expr_ << " evaluates to " << last_expr_ << std::endl; \
    throw test_error(); \
  } \
} while (0)

ASSERT(i == j)
do { \
  if (!(Capture()->*i == j)) { \
    std::cerr << ... << "i == j" << " evaluates to " << last_expr_ << std::endl; \
    throw test_error(); \
  } \
} while (0)
```

Capture

```
Capture()->*i == j
```

Capture

```
Capture()->*i == j
Capture()->*(i) == j
```

Capture

```
Capture()->*i == j
Capture()->*(i) == j
```

```
struct Capture {
  template<typename T>
  Term<T> operator->*(const T& term) {
    return Term<T>(term);
  }
};
```

Term

```
Capture()->*i == j
Capture()->*(i) == j
```

```
Term<int>(i) == j
```

```
struct Capture {
  template<typename T>
  Term<T> operator->*(const T& term) {
    return Term<T>(term);
  }
};
```

Term

```
Term<int>(i) == j
```

```
template<typename T>
class Term {
  const T& lhs_;
public:
  Term(const T& lhs) : lhs_(lhs) { }
};
```

Term

```
Term<int>(i) == j
```

```
template<typename T>
class Term {
  const T& lhs_;
public:
  Term(const T& lhs) : lhs_(lhs) { }
  template<typename U>
  bool operator==(const U& rhs) const {
    // lhs_ == i, rhs == j
    // ...


  }
};
```

Term

```
Term<int>(i) == j
```

```
template<typename T>
class Term {
  const T& lhs_;
public:
  Term(const T& lhs) : lhs_(lhs) { }
  template<typename U>
  bool operator==(const U& rhs) const {
    std::ostringstream oss;
    oss << lhs_ << " == " << rhs;
    last_expr_ = oss.str();
    // ...
  }
};
```

Term

```
Term<int>(i) == j
```

```
template<typename T>
class Term {
  const T& lhs_;
public:
  Term(const T& lhs) : lhs_(lhs) { }
  template<typename U>
  bool operator==(const U& rhs) const {
    std::ostringstream oss;
    oss << lhs_ << " == " << rhs;
    last_expr_ = oss.str();
    return lhs_ == rhs;
  }
};
```

Term

```
Term<int>(i) == j
```

```
template<typename T>
class Term {
  const T& lhs_;
public:
  Term(const T& lhs) : lhs_(lhs) { }
  template<typename U>
  bool operator==(const U& rhs) const {
    std::ostringstream oss;
    oss << lhs_ << " == " << rhs;
    last_expr_ = oss.str();
    return lhs_ == rhs;
  }
};
```

Test monolitico

Test monolitico

```
#include <gut.h>
#include <string-stack.h>

TEST_BEGIN

  StringStack stack = StringStack();
  CHECK(stack.empty());

  std::string aString = "Only String";
  stack.push(aString);
  CHECK(!stack.empty());

  std::string topValue = stack.top();
  CHECK(aString == topValue);
  CHECK(!stack.empty());

  stack.pop();
  CHECK(stack.empty());

TEST_END
```

Test case

```
#include <gut.h>
#include <string-stack.h>
```

Specifiche

```
#include <gut.h>
#include <string-stack.h>

TEST("initial stack is empty") {
  StringStack anEmptyStack;
  CHECK(anEmptyStack.empty());
}
```

Specifiche

```
#include <gut.h>
#include <string-stack.h>

TEST("initial stack is empty") {
  StringStack anEmptyStack;
  CHECK(anEmptyStack.empty());
}

TEST("items are extracted in last-in-first-out order") {
  StringStack aStackWithManyElements;
  aStackWithManyElements.push("one");
  aStackWithManyElements.push("two");

  CHECK(aStackWithManyElements.top() == "two");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.top() == "one");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.empty());
}
```

Specifiche

```
#include <gut.h>
#include <string-stack.h>

TEST("initial stack is empty") {
  StringStack anEmptyStack;
  CHECK(anEmptyStack.empty());
}

TEST("items are extracted in last-in-first-out order") {
  StringStack aStackWithManyElements;
  aStackWithManyElements.push("one");
  aStackWithManyElements.push("two");

  CHECK(aStackWithManyElements.top() == "two");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.top() == "one");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.empty());
}
```

AAA /arrange

```
#include <gut.h>
#include <string-stack.h>

TEST("initial stack is empty") {
  StringStack anEmptyStack;
  CHECK(anEmptyStack.empty());
}

TEST("items are extracted in last-in-first-out order") {
  StringStack aStackWithManyElements;
  aStackWithManyElements.push("one");
  aStackWithManyElements.push("two");

  CHECK(aStackWithManyElements.top() == "two");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.top() == "one");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.empty());
}
```

AAA /act

```
#include <gut.h>
#include <string-stack.h>

TEST("initial stack is empty") {
  StringStack anEmptyStack;
  CHECK(anEmptyStack.empty());
}

TEST("items are extracted in last-in-first-out order") {
  StringStack aStackWithManyElements;
  aStackWithManyElements.push("one");
  aStackWithManyElements.push("two");

  CHECK(aStackWithManyElements.top() == "two");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.top() == "one");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.empty());
}
```

AAA /assert

```
#include <gut.h>
#include <string-stack.h>

TEST("initial stack is empty") {
  StringStack anEmptyStack;
  CHECK(anEmptyStack.empty());
}

TEST("items are extracted in last-in-first-out order") {
  StringStack aStackWithManyElements;
  aStackWithManyElements.push("one");
  aStackWithManyElements.push("two");

  CHECK(aStackWithManyElements.top() == "two");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.top() == "one");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.empty());
}
```

Specifiche

```
#include <gut.h>
#include <string-stack.h>

// ...

TEST("top called on an empty stack throws an exception") {
  StringStack anEmptyStack;
  THROWS(anEmptyStack.top(), stack_empty);
}

TEST("pop called on an empty stack throws an exception") {
  StringStack anEmptyStack;
  THROWS(anEmptyStack.pop(), stack_empty);
}
```

Specifiche

```
#include <gut.h>
#include <string-stack.h>

// ...
```

```
Test suite started...
initial stack is empty: OK
items are extracted in last-in-first-out order: OK
top called on an empty stack throws an exception: OK
pop called on an empty stack throws an exception: OK
Ran 4 test(s) in 0.002 s.
OK - all tests passed.
```

Specifiche

```
#include <gut.h>
#include <string-stack.h>

// ...
```

```
Test suite started...
initial stack is empty: OK
items are extracted in last-in-first-out order: FAILED
 C:\...\stack.cpp(30) : [error] aStackWithManyElements.top() == "two" evaluates to "one" == "two"
 C:\...\stack.cpp(32) : [error] aStackWithManyElements.top() == "one" evaluates to "two" == "one"
top called on an empty stack throws an exception: OK
pop called on an empty stack throws an exception: OK
Ran 4 test(s) in 0.001 s.
FAILED - 2 failure(s) in 1 test(s).
```

Test procedurale

```
#include <gut.h>
#include <string-stack.h>

TEST("empty") {
  StringStack aStack;
  CHECK(aStack.empty());

  aStack.push("one");
  CHECK(!aStack.empty());

  aStack.pop();
  CHECK(aStack.empty());
}

TEST("top") {
  // ...
}
```

Test procedurale

```
#include <gut.h>
#include <string-stack.h>

TEST("empty") {
  StringStack aStack;
  CHECK(aStack.empty());

  aStack.push("one"); // using push to test empty!
  CHECK(!aStack.empty());

  aStack.pop();       // using pop to test empty!
  CHECK(aStack.empty());
}

TEST("top") {
  // ...
}
```

TEST

```
#include ...
#include <gut.h>

TEST("a test") {
  // ...
}
```

TestFn

```
typedef void (*TestFn)();
```

Test

```
typedef void (*TestFn)();

class Test {
  std::string name_;
  TestFn test_;
public:
  Test(const std::string& name, TestFn test) : name_(name), test_(test) { }
  const std::string& name() const { return name_; }
  void run() { test_(); }
};
```

Test, TestSuite

```
typedef void (*TestFn)();

class Test {
  std::string name_;
  TestFn test_;
public:
  Test(const std::string& name, TestFn test) : name_(name), test_(test) { }
  const std::string& name() const { return name_; }
  void run() { test_(); }
};

struct TestSuite {
  static std::vector<Test> tests_;
  struct add {
    add(const std::string& name, TestFn test) { tests_.push_back(Test(name, test)); }
  };
};
```

TEST

```
#define TEST(name_) \
  static void MAKE_UNIQUE(test_)(); \
  gut::TestSuite::add MAKE_UNIQUE(testAddition_)(name_, &CONCAT_(test_, __LINE__)); \
  static void MAKE_UNIQUE(test_)()
```

TEST

```
#define TEST(name_) \
  static void MAKE_UNIQUE(test_)(); \
  gut::TestSuite::add MAKE_UNIQUE(testAddition_)(name_, &CONCAT_(test_, __LINE__)); \
  static void MAKE_UNIQUE(test_)()

TEST("a test") {
  // test body
}
```

TEST

```
#define TEST(name_) \
  static void MAKE_UNIQUE(test_)(); \
  gut::TestSuite::add MAKE_UNIQUE(testAddition_)(name_, &CONCAT_(test_, __LINE__)); \
  static void MAKE_UNIQUE(test_)()

static void test_123();
gut::TestSuite::add testAddition_123("a test", test_123);
static void test_123() {
  // test body
}
```

TEST

```
#define TEST(name_) \
  static void MAKE_UNIQUE(test_)(); \
  gut::TestSuite::add MAKE_UNIQUE(testAddition_)(name_, &CONCAT_(test_, __LINE__)); \
  static void MAKE_UNIQUE(test_)()

static void test_123();
gut::TestSuite::add testAddition_123("a test", test_123);
static void test_123() {
  // test body
}
```

TEST

```
#define TEST(name_) \
  static void MAKE_UNIQUE(test_)(); \
  gut::TestSuite::add MAKE_UNIQUE(testAddition_)(name_, &CONCAT_(test_, __LINE__)); \
  static void MAKE_UNIQUE(test_)()

static void test_123();
gut::TestSuite::add testAddition_123("a test", test_123);
static void test_123() {
  // test body
}
```

TEST

```
#define TEST(name_) \
  static void MAKE_UNIQUE(test_)(); \
  gut::TestSuite::add MAKE_UNIQUE(testAddition_)(name_, &CONCAT_(test_, __LINE__)); \
  static void MAKE_UNIQUE(test_)()

static void test_123();
gut::TestSuite::add testAddition_123("a test", test_123);
static void test_123() {
  // test body
}
```

main

```
int main() {
  return runTests_();
}
```

main

```
int runTests_() {
  for (auto test : gut::TestSuite::tests()) {
    try {
      test.run();
    } catch(...) {
      // ...
    }
  }
  return failedTestCount;
}

int main() {
  return runTests_();
}
```

Caratteristiche principali

Macro

Macro

- Asserzioni

Macro

- Asserzioni
  - **`ASSERT(expr);`**

Macro

- Asserzioni
  - `ASSERT(expr);  // non bloccante`

Macro

- Asserzioni
  - `CHECK(expr);   // non bloccante`

Macro

- Asserzioni
  - `CHECK(expr);   // non bloccante`
  - `REQUIRE(expr); // bloccante`

Macro

- Asserzioni
  - `CHECK(expr);`
  - `REQUIRE(expr);`
- Eccezioni

Macro

- Asserzioni
  - `CHECK(expr);`
  - `REQUIRE(expr);`
- Eccezioni
  - `THROWS(expr, type);`

Macro

- Asserzioni
  - `CHECK(expr);`
  - `REQUIRE(expr);`
- Eccezioni
  - `THROWS(expr, type);`
  - `THROWS_WITH_MESSAGE(expr, type, what);`

Macro

- Asserzioni
  - `CHECK(expr);`
  - `REQUIRE(expr);`
- Eccezioni
  - `THROWS(expr, type);`
  - `THROWS_WITH_MESSAGE(expr, type, what);`
  - `THROWS_ANYTHING(expr);`

Macro

- Asserzioni
  - `CHECK(expr);`
  - `REQUIRE(expr);`
- Eccezioni
  - `THROWS(expr, type);`
  - `THROWS_WITH_MESSAGE(expr, type, what);`
  - `THROWS_ANYTHING(expr);`
  - `THROWS_NOTHING(expr);`

Macro

- Asserzioni
  - `CHECK(expr);`
  - `REQUIRE(expr);`
- Eccezioni
  - `[REQUIRE_]THROWS(expr, type);`
  - `[REQUIRE_]THROWS_WITH_MESSAGE(expr, type, what);`
  - `[REQUIRE_]THROWS_ANYTHING(expr);`
  - `[REQUIRE_]THROWS_NOTHING(expr);`

Macro

- Messaggi

Macro

- Messaggi
  - `EVAL(expr);`

Macro

- Messaggi
  - `EVAL(expr);     // shows only if test fails`

Macro

- Messaggi
  - `EVAL(expr);     // shows only if test fails`
  - `INFO(message);`

Macro

- Messaggi
  - `EVAL(expr);      // shows only if test fails`
  - `INFO(message);   // shows only if test fails`

Macro

- Messaggi
  - `EVAL(expr);      // shows only if test fails`
  - `INFO(message);   // shows only if test fails`
  - `WARN(message);`

Macro

- Messaggi
  - `EVAL(expr);      // shows only if test fails`
  - `INFO(message);   // shows only if test fails`
  - `WARN(message);`
  - `FAIL(message);`

Macro

- Messaggi
  - `EVAL(expr);      // shows only if test fails`
  - `INFO(message);   // shows only if test fails`
  - `WARN(message);`
  - `FAIL(message);   // causes the test to fail`

Configurabilità

Configurabilità

- **`GUT_ENABLE_FAILFAST`**

Configurabilità

- `GUT_ENABLE_FAILFAST`
- `GUT_CUSTOM_REPORT(myReport)`

Configurabilità

- `GUT_ENABLE_FAILFAST`
- `GUT_CUSTOM_REPORT(myReport)`
- `#define GUT_CUSTOM_MAIN`

Configurabilità

- **GUT_ENABLE_FAILFAST**
- **GUT_CUSTOM_REPORT(myReport)**
- **#define GUT_CUSTOM_MAIN**

```
int main() {
  // some stuff...
  runTests_();
  // other stuff...
}
```

GUT vs googletest

GUT vs googletest

```
#include <gut.h>
#include <string-stack.h>

TEST("initial stack is empty") {
  StringStack anEmptyStack;
  CHECK(anEmptyStack.empty());
}

TEST("items are extracted in last-in-first-out order") {
  StringStack aStackWithManyElements;
  aStackWithManyElements.push("one");
  aStackWithManyElements.push("two");

  CHECK(aStackWithManyElements.top() == "two");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.top() == "one");
  aStackWithManyElements.pop();
  CHECK(aStackWithManyElements.empty());
}
```

GUT vs **googletest**

```
#include <gtest.h>
#include <string-stack.h>

TEST(StringStack, InitialStackIsEmpty) {
  StringStack anEmptyStack;
  EXPECT_TRUE(anEmptyStack.empty());
}

TEST(StringStack, ItemsAreExtractedInLIFOOrder) {
  StringStack aStackWithManyElements;
  aStackWithManyElements.push("one");
  aStackWithManyElements.push("two");

  EXPECT_EQ(aStackWithManyElements.top(), "two");
  aStackWithManyElements.pop();
  EXPECT_EQ(aStackWithManyElements.top(), "one");
  aStackWithManyElements.pop();
  EXPECT_TRUE(aStackWithManyElements.empty());
}
```

GUT vs googletest

```
// ...

TEST("top called on an empty stack throws an exception") {
  StringStack anEmptyStack;
  THROWS(anEmptyStack.top(), stack_empty);
}

TEST("pop called on an empty stack throws an exception") {
  StringStack anEmptyStack;
  THROWS(anEmptyStack.pop(), stack_empty);
}
```

GUT vs **googletest**

```
// ...

TEST(StringStack, TopCalledOnAnEmptyStackThrowsAnException) {
  StringStack anEmptyStack;
  EXPECT_THROW(anEmptyStack.top(), stack_empty);
}

TEST(StringStack, PopCalledOnAnEmptyStackThrowsAnException) {
  StringStack anEmptyStack;
  EXPECT_THROW(anEmptyStack.pop(), stack_empty);
}
```

GUT vs **googletest**

```
// ...

TEST(StringStack, TopCalledOnAnEmptyStackThrowsAnException) {
  StringStack anEmptyStack;
  EXPECT_THROW(anEmptyStack.top(), stack_empty);
}

TEST(StringStack, PopCalledOnAnEmptyStackThrowsAnException) {
  StringStack anEmptyStack;
  EXPECT_THROW(anEmptyStack.pop(), stack_empty);
}

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

GUT vs googletest

```
Test suite started...
initial stack is empty: OK
items are extracted in last-in-first-out order: OK
top called on an empty stack throws an exception: OK
pop called on an empty stack throws an exception: OK
Ran 4 test(s) in 0.001s.
OK - all tests passed.
```

GUT vs **googletest**

```
[==========] Running 4 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 4 tests from StringStack
[ RUN      ] StringStack.InitialStackIsEmpty
[       OK ] StringStack.InitialStackIsEmpty (0 ms)
[ RUN      ] StringStack.ItemsAreExtractedInLIFOOrder
[       OK ] StringStack.ItemsAreExtractedInLIFOOrder (0 ms)
[ RUN      ] StringStack.TopCalledOnAnEmptyStackThrowsAnException
[       OK ] StringStack.TopCalledOnAnEmptyStackThrowsAnException (0 ms)
[ RUN      ] StringStack.PopCalledOnAnEmptyStackThrowsAnException
[       OK ] StringStack.PopCalledOnAnEmptyStackThrowsAnException (0 ms)
[----------] 4 tests from StringStack (29 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (43 ms total)
[  PASSED  ] 4 tests.
```

GUT vs googletest

GUT vs googletest

(-) Necessita del link di una libreria statica

GUT vs googletest

(-) Necessita del link di una libreria statica

(-) Non consente la verifica diretta del `what` delle eccezioni

GUT vs googletest

GUT vs googletest

(+) Shuffling dei test

GUT vs googletest

(+) Shuffling dei test

(+) Ripetizione ciclica dei test

GUT vs googletest

(+) Shuffling dei test

(+) Ripetizione ciclica dei test

(+) *Break-on-failure*

GUT vs googletest

(+) Shuffling dei test

(+) Ripetizione ciclica dei test

(+) *Break-on-failure*

(+) Supporto dei *death-test*

GUT vs googletest

GUT vs googletest

(?) Più test-case in un unico file

GUT vs googletest

(?) Più test-case in un unico file

(?) Selezione del test da eseguire

GUT vs googletest

(?) Più test-case in un unico file

(?) Selezione del test da eseguire

(?) Supporto delle *fixtures*

GUT vs googletest

(?) Più test-case in un unico file

(?) Selezione del test da eseguire

(?) Supporto delle *fixtures*

(?) Test parametrici su tipi e valori

GUT vs googletest

(?) Più test-case in un unico file

(?) Selezione del test da eseguire

(?) Supporto delle *fixtures*

(?) Test parametrici su tipi e valori

(?) Test *listeners*

Grazie!