
www.italiancpp.org

# Sviluppo di un framework di unit-test C++

Gianfranco Zuliani

Bologna, 8 Novembre 2014

# Origini

# Origini

- Primi anni 2000

# Origini

- Primi anni 2000
- Processo waterfall

# Origini

- Primi anni 2000
- Processo waterfall
- Il progetto fallisce

# Origini

- Primi anni 2000
- Processo waterfall
- Il progetto fallisce
- Un cliente decide di riprovarci

# Desiderata

# Desiderata

*Un sistema di test automatici!*

# Desiderata

*Un sistema di test automatici:*

- Portabile

# Desiderata

*Un sistema di test automatici:*

- Portabile (Windows/Linux, ...)

# Desiderata

*Un sistema di test automatici:*

- Portabile (Windows/Linux, VC/MinGW/GCC, ...)

# Desiderata

*Un sistema di test automatici:*

- Portabile (Windows/Linux, VC/MinGW/GCC, x86/amd64, ...)

# Desiderata

*Un sistema di test automatici:*

- Portabile
- Impatto minimo

# Desiderata

*Un sistema di test automatici:*

- Portabile
- Impatto minimo (no ERROR, ...)

# Desiderata

*Un sistema di test automatici:*

- Portabile
- Impatto minimo (no ERROR, /MD vs. /MT, ...)

# Desiderata

*Un sistema di test automatici:*

- Portabile
- Impatto minimo
- Gestione delle eccezioni

# Desiderata

*Un sistema di test automatici:*

- Portabile
- Impatto minimo
- Gestione delle eccezioni (no errno, ...)

# Desiderata

*Un sistema di test automatici:*

- Portabile
- Impatto minimo
- Gestione delle eccezioni (no errno, HRESULT, ...)

# Desiderata

*Un sistema di test automatici:*

- Portabile
- Impatto minimo
- Gestione delle eccezioni
- No macro differenziate

# Desiderata

*Un sistema di test automatici:*

- Portabile
- Impatto minimo
- Gestione delle eccezioni
- No macro differenziate (EQUAL vs. EQUAL_DBL vs. EQUAL_STR ...)

# Desiderata

*Un sistema di test automatici:*

- Portabile
- Impatto minimo
- Gestione delle eccezioni
- No macro differenziate
- Output configurabile

# Al tempo...

# Al tempo...

- Esperienza sul testing automatico pressoché nulla

# Al tempo...

- Esperienza sul testing automatico pressoché nulla
- Idee non molto chiare su cosa sarebbe servito

# Al tempo...

- Esperienza sul testing automatico pressoché nulla
- Idee non molto chiare su cosa sarebbe servito
- Molti aspetti del testing non ancora ben formalizzati

# Al tempo...

- Esperienza sul testing automatico pressoché nulla
- Idee non molto chiare su cosa sarebbe servito
- Molti aspetti del testing non ancora ben formalizzati
- Assenza di una soluzione adottata su larga scala

# La soluzione

# La soluzione

***Farselo in casa!***

# Test file

```
#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

# ASSERT /2

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

TEST_BEGIN

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


TEST_END
```

# ASSERT /2

```
#include ...
#include <gut.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;
  }
};
```

# compare

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

TEST_BEGIN

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

TEST_END
```

# compare

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

TEST_BEGIN

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

TEST_END
```

*warning: comparison between signed and unsigned integer expressions*

# compare

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

# compare

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

# compare

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

# compare

```
template<typename T>
class Term {
  // ...
  template<typename U>
  bool operator==(const U& rhs) const {
    return compare(lhs_, rhs);
  }
};
```

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

# compare

```
template<typename T>
class Term {
  // ...
  template<typename U>
  bool operator==(const U& rhs) const {
    return compare(lhs_, rhs);
  }
};
```

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

bool compare(unsigned lhs, int rhs) {
  return compare(
    static_cast<long>(lhs),
    static_cast<long>(rhs));
}
```

# operator<

```
template<typename T>
class Term {
  // ...
  template<typename U>
  bool operator==(const U& rhs) const {
    return compare(lhs_, rhs);
  }
  template<typename U>
  bool operator<(const U& rhs) const {
    return ???
  }
};
```

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

bool compare(unsigned lhs, int rhs) {
  return compare(
    static_cast<long>(lhs),
    static_cast<long>(rhs));
}
```

# operator<

```
template<typename T>
class Term {
  // ...
  template<typename U>
  bool operator==(const U& rhs) const {
    return compare<e_equal>(lhs_, rhs);
  }
  template<typename U>
  bool operator<(const U& rhs) const {
    return compare<e_lessThan>(lhs_, rhs);
  }
};
```

```
enum Operator { e_equal, e_lessThan, ... };

template<Operator op, typename T, typename U>
bool compare(const T& lhs, const U& rhs) {
  return ExprFactory<T, U, op>::logAndEval(lhs, rhs);
}
```

```
template<Operator op>
bool compare(unsigned lhs, int rhs) {
  return compare<op>(
    static_cast<long>(lhs),
    static_cast<long>(rhs));
}
```

# ExprFactory

```
struct OPERATION_NOT_SUPPORTED;

template<typename T, typename U, Operator op>
struct ExprFactory {
  static bool logAndEval(const T&, const U&) { return OPERATION_NOT_SUPPORTED(); }
};
```

# ExprFactory

```
struct OPERATION_NOT_SUPPORTED;

template<typename T, typename U, Operator op>
struct ExprFactory {
  static bool logAndEval(const T&, const U&) { return OPERATION_NOT_SUPPORTED(); }
};

template<typename T, typename U>
struct ExprFactory<T, U, e_equal> {
  static bool logAndEval(const T& lhs, const U& rhs) {
    return Equal<T, U>(lhs, rhs).logAndEval();
  }
};
```

# ExprFactory

```
struct OPERATION_NOT_SUPPORTED;

template<typename T, typename U, Operator op>
struct ExprFactory {
  static bool logAndEval(const T&, const U&) { return OPERATION_NOT_SUPPORTED(); }
};

template<typename T, typename U>
struct ExprFactory<T, U, e_equal> {
  static bool logAndEval(const T& lhs, const U& rhs) {
    return Equal<T, U>(lhs, rhs).logAndEval();
  }
};

template<typename T, typename U>
struct ExprFactory<T, U, e_lessThan> {
  static bool logAndEval(const T& lhs, const U& rhs) {
    return LessThan<T, U>(lhs, rhs).logAndEval();
  }
};
```

# Expr

```
struct Expr {
  bool logAndEval() {
      last_expr_ = toStr();
      return eval();
  }


};
```

# Expr

```
struct Expr {
  bool logAndEval() {
      last_expr_ = toStr();
      return eval();
  }
  virtual std::string toStr() const = 0;
  virtual bool eval() const = 0;
};
```

# BinaryExpr

```
struct Expr {
  bool logAndEval() {
      last_expr_ = toStr();
      return eval();
  }
  virtual std::string toStr() const = 0;
  virtual bool eval() const = 0;
};

template<typename T, typename U>
class BinaryExpr : public Expr {
protected:
    const T& lhs_;
    const U& rhs_;
public:
    BinaryExpr(const T& lhs, const U& rhs) : lhs_(lhs), rhs_(rhs) { }
    std::string toStr() const override { return toStr(lhs_) + " " + opRepr() + " " + toStr(rhs_); }

};
```

# BinaryExpr

```
struct Expr {
  bool logAndEval() {
      last_expr_ = toStr();
      return eval();
  }
  virtual std::string toStr() const = 0;
  virtual bool eval() const = 0;
};

template<typename T, typename U>
class BinaryExpr : public Expr {
protected:
    const T& lhs_;
    const U& rhs_;
public:
    BinaryExpr(const T& lhs, const U& rhs) : lhs_(lhs), rhs_(rhs) { }
    std::string toStr() const override { return toStr(lhs_) + " " + opStr() + " " + toStr(rhs_); }
    virtual std::string opStr() const = 0;
};
```

# Equal

```
template<typename T, typename U>
struct Equal : public BinaryExpression<T, U> {
  Equal(const T& lhs, const U& rhs) : BinaryExpr<T, U>(lhs, rhs) { }


};
```

# Equal

```
template<typename T, typename U>
struct Equal : public BinaryExpression<T, U> {
  Equal(const T& lhs, const U& rhs) : BinaryExpr<T, U>(lhs, rhs) { }
  bool eval() const override { return this->lhs_ == this->rhs_; }
  std::string opStr() const override { return "=="; }
};
```

# LessThan

```
template<typename T, typename U>
struct Equal : public BinaryExpression<T, U> {
  Equal(const T& lhs, const U& rhs) : BinaryExpr<T, U>(lhs, rhs) { }
  bool eval() const override { return this->lhs_ == this->rhs_; }
  std::string opStr() const override { return "=="; }
};

template<typename T, typename U>
struct LessThan : public BinaryExpr<T, U> {
  LessThan(const T& lhs, const U& rhs) : BinaryExpr<T, U>(lhs, rhs) { }


};
```

# LessThan

```
template<typename T, typename U>
struct Equal : public BinaryExpression<T, U> {
  Equal(const T& lhs, const U& rhs) : BinaryExpr<T, U>(lhs, rhs) { }
  bool eval() const override { return this->lhs_ == this->rhs_; }
  std::string getOp() const override { return "=="; }
};

template<typename T, typename U>
struct LessThan : public BinaryExpr<T, U> {
  LessThan(const T& lhs, const U& rhs) : BinaryExpr<T, U>(lhs, rhs) { }
  virtual bool evaluate() const { return this->lhs_ < this->rhs_; }
  virtual std::string opStr() const { return "<"; }
};
```

# toStr

```
template<typename T>
std::string toStr(const T& value) {
  std::ostringstream os;
  os << std::boolalpha << value;
  return os.str();
}
```

# toStr

```
template<typename T>
std::string toStr(const T& value) {
  std::ostringstream os;
  os << std::boolalpha << value;
  return os.str();
}

std::string toStr(const std::string& value) {
  return std::string("\"") + value + "\"";
}
```

# toStr

```
template<typename T>
std::string toStr(const T& value) {
  std::ostringstream os;
  os << std::boolalpha << value;
  return os.str();
}

std::string toStr(const std::string& value) {
  return std::string("\"") + value + "\"";
}

std::string toString(std::nullptr_t) {
    return "<nullptr>";
}
```

# toStr

```
template<typename T>
std::string toStr(const T& value) {
  std::ostringstream os;
  os << std::boolalpha << value;
  return os.str();
}

std::string toStr(const std::string& value) {
  return std::string("\"") + value + "\"";
}

std::string toString(std::nullptr_t) {
    return "<nullptr>";
}

// ...
```

# Strategia

# Strategia

- Catturare il primo termine

# Strategia

- Catturare il primo termine (`Term::Term`)

# Strategia

- Catturare il primo termine (`Term::Term`)
- Catturare il secondo termine

# Strategia

- Catturare il primo termine (`Term::Term`)
- Catturare il secondo termine (`Term::operator==` & Co.)

# Strategia

- Catturare il primo termine (`Term::Term`)
- Catturare il secondo termine (`Term::operator==` & Co.)
- Adattare i tipi

# Strategia

- Catturare il primo termine (`Term::Term`)
- Catturare il secondo termine (`Term::operator==` & Co.)
- Adattare i tipi (`compare`)

# Strategia

- Catturare il primo termine (`Term::Term`)
- Catturare il secondo termine (`Term::operator==` & Co.)
- Adattare i tipi (`compare`)
- Istanziare l'espressione

# Strategia

- Catturare il primo termine (**`Term::Term`**)
- Catturare il secondo termine (**`Term::operator==`** & Co.)
- Adattare i tipi (**`compare`**)
- Istanziare l'espressione (**`ExprFactory`**)

# Strategia

- Catturare il primo termine (`Term::Term`)
- Catturare il secondo termine (`Term::operator==` & Co.)
- Adattare i tipi (`compare`)
- Istanziare l'espressione (`ExprFactory`)
- Serializzare e valutare l'espressione

# Strategia

- Catturare il primo termine (`Term::Term`)
- Catturare il secondo termine (`Term::operator==` & Co.)
- Adattare i tipi (`compare`)
- Istanziare l'espressione (`ExprFactory`)
- Serializzare e valutare l'espressione (`Expr`, `toStr`)

# 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(...) {
      // log error
    }

  }

  return failedTestCount;
}

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

# Prospetti

```
int runTests_() {
  theReport.start();
  for (auto test : gut::TestSuite::tests()) {
    theReport.start_test(test.name());
    try {
      test.run();
    } catch(...) {
      theReport.failure(...);
    }
    theReport.endTest();
  }
  theReport.end();
  return failedTestCount;
}

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

# Prospetti

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

# Prospetti

```
#define CHECK(expr_) \
do { \
  if (!(Capture()->*expr_)) { \
    theReport.failure(__FILE__ ... __LINE__ ... #expr_ ... last_expr_); \
    throw test_error(); \
  } \
} while (0)
```

# Prospetti

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



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

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

# Prospetti

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

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

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

```
Test suite started...
a test: OK
another test: OK
Ran 2 test(s) in 0.001 s.
OK - all tests passed.
```

# Report

```
#include ...
#include <gut.h>
#include <tap-report.h>

GUT_ENABLE_REPORT(TapReport())

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

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

# Report

```
#include ...
#include <gut.h>
#include <tap-report.h>

GUT_ENABLE_REPORT(TapReport())

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

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

```
ok 1 - a test
ok 2 - another test
1..2
# failed 0/5 test(s), 100.0% ok
```

# Report

```
// gut.h

struct Report {
  virtual void failure(const std::string& message) = 0;
  // ...
};
```

# Report

```
// gut.h

struct Report {
  virtual void failure(const std::string& message) = 0;
  // ...
};

struct DefaultReport : Report {
  virtual void failure(const std::string& message) { // ... }
  // ...
};
```

# Report

```
// gut.h

struct Report {
  virtual void failure(const std::string& message) = 0;
  // ...
};

struct DefaultReport : Report {
  virtual void failure(const std::string& message) { // ... }
  // ...
};

std::auto_ptr<Report> theReport(new DefaultReport());
```

# Report

```
// tap-report.h
#include <gut.h>

struct TapReport : gut::Report {
  virtual void failure(const std::string& message);
  // ...
};
```

# Report

```
// tap-report.h
#include <gut.h>

struct TapReport : gut::Report {
  virtual void failure(const std::string& message);
  // ...
};
```

```
// my-test.cpp
gut::theReport.reset(new TapReport());
```

# Report

```
// tap-report.h
#include <gut.h>

struct TapReport : gut::Report {
  virtual void failure(const std::string& message);
  // ...
};


// my-test.cpp
gut::theReport.reset(new TapReport());
```

# Type Erasure /1

```
struct TapReport {
  void failure(const std::string& message);
  // ...
};
```

# Type Erasure /1

```
struct TapReport {
  void failure(const std::string& message);
  // ...
};

// TapReport myReport;
```

# Type Erasure /1

```
struct TapReport {
  void failure(const std::string& message);
  // ...
};

template<typename T>
struct Model {
  T report_;
  Model(T report) : report_(report) { }
  void failure(const std::string& message) { report_.failure(message); }
  // ...
};
```

# Type Erasure /1

```
struct TapReport {
  void failure(const std::string& message);
  // ...
};

template<typename T>
struct Model {
  T report_;
  Model(T report) : report_(report) { }
  void failure(const std::string& message) { report_.failure(message); }
  // ...
};

// Model<TapReport> myReport;
```

# Type Erasure /2

```
struct TapReport {
  void failure(const std::string& message);
  // ...
};

struct Concept {
  virtual void failure(const std::string& message) = 0;
};

template<typename T>
struct Model : Concept {
  T report_;
  Model(T report) : report_(report) { }
  void failure(const std::string& message) override { report_.failure(message); }
  // ...
};
```

# Type Erasure /2

```
struct TapReport {
  void failure(const std::string& message);
  // ...
};

struct Concept {
  virtual void failure(const std::string& message) = 0;
};

template<typename T>
struct Model : Concept {
  T report_;
  Model(T report) : report_(report) { }
  void failure(const std::string& message) override { report_.failure(message); }
  // ...
};

// std::shared_ptr<Concept> myReport = std::make_shared<Model<TapReport>>();
```

# Type Erasure /3

```
struct Report {
  std::shared_ptr<Concept> report_;
  template<class T>
  Report(T report) : report_(std::make_shared<Model<T>>(report)) { }
  void failure(const std::string& message) { report_->failure(message); }
};
```

# Type Erasure /3

```
struct Report {
  std::shared_ptr<Concept> report_;
  template<class T>
  Report(T report) : report_(std::make_shared<Model<T>>(report)) { }
  void failure(const std::string& message) { report_->failure(message); }
};

// gut.h
Report theReport((DefaultReport()));
```

# Type Erasure /3

```
struct Report {
  std::shared_ptr<Concept> report_;
  template<class T>
  Report(T report) : report_(std::make_shared<Model<T>>(report)) { }
  void failure(const std::string& message) { report_->failure(message); }
};

// gut.h
Report theReport((DefaultReport()));

// my-test.cpp
#include <tap-report.h>
theReport = Report((TapReport()));
```

# Type Erasure /3

```
struct Report {
  std::shared_ptr<Concept> report_;
  template<class T>
  Report(T report) : report_(std::make_shared<Model<T>>(report)) { }
  void failure(const std::string& message) { report_->failure(message); }
};

// gut.h
Report theReport((DefaultReport()));

// my-test.cpp
#include <tap-report.h>
GUT_ENABLE_REPORT(TapReport);
```

# *Caratteristiche principali*

# Macro

# Macro

- Asserzioni

# Macro

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

# Macro

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

# 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_REPORT(myReport);`

# Configurabilità

- `GUT_ENABLE_REPORT(myReport);`
- `GUT_ENABLE_FAILFAST`

# Configurabilità

- `GUT_ENABLE_REPORT(myReport);`
- `GUT_ENABLE_FAILFAST`
- `#define GUT_CUSTOM_MAIN`

# Configurabilità

- `GUT_ENABLE_REPORT(myReport);`
- `GUT_ENABLE_FAILFAST`
- `#define GUT_CUSTOM_MAIN`

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

# Configurabilità

- `GUT_ENABLE_REPORT(myReport);`
- `GUT_ENABLE_FAILFAST`
- `#define GUT_CUSTOM_MAIN`
- `#define INT_BASE Dec|Hex`

# 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!