Google Test & Mock Platform

Complete Tutorial

Part 2: GMock Matchers

By Ari Saif

```
TEST(MatchersTest, EXPECT_THAT) {
  EXPECT_THAT(1, 1);
  EXPECT_THAT(2 - 2, 0);
  EXPECT_THAT(1 == 1, true);
}
```

EXPECT_THAT

```
TEST(MatchersTest, ContainerMatchers) {
  std::vector<int> v = {4, 6, 8, 2, 2};
  EXPECT_THAT(v, Each(Gt(1)));
  EXPECT_THAT(v, Contains(6));
  EXPECT_THAT(v, Contains(Gt(1)));
  EXPECT_THAT(v, Each(AnyOf(Gt(2), Gt(0))));
}
```

Container Matchers

```
MATCHER_P2(IsBetween, a, b,
    (std::string(negation ? "Isn't" : "Is")
    + " between "
    + PrintToString(a) + " and " + PrintToString(b))) {
  return a <= arg && arg <= b;
}

TEST(MatchersTest, CustomMatchers_WithDescription) {
  EXPECT_THAT(100, IsBetween(9, 11));
  EXPECT_THAT(10, Not(IsBetween(9, 11)));
}
```

Costume Matchers

```
TEST(MatchersTest, StringMatchers) {
  auto str1 = std::string("This is aa test. Thanks for watching.");
  EXPECT_THAT(str1, ContainsRegex("[a]+ test\\."));
  EXPECT_THAT(str1, MatchesRegex("T.*\\."));
  EXPECT_THAT(str1, EndsWith("watching."));
  EXPECT_THAT(str1, HasSubstr("Thanks for"));
}
```

String Matchers

Matcher
AllOf(m1, m2, ..., mn)
AllOfArray({m0, m1, ..., mn}), AllOfArray(a_container), AllOfArray(begin, end), AllOfArray(array), or AllOfArray(array, count)
AnyOf(m1, m2, ..., mn)
AnyOfArray({m0, m1, ..., mn}), AnyOfArray(a_container), AnyOfArray(begin, end), AnyOfArray(array), or AnyOfArray(array, count)
Not(m)
Conditional(cond, m1, m2)

Composite Matchers

What Are We Covering?

- Google **Test**
  - Google Test Macros
  - Test **Fixtures**
  - Manipulating Tests
  - Advanced Topics
    - Sharing Test Logic
    - Typed Test
    - Type-Parameterized Test
    - Assertion in Subroutines
    - Testing Private Code

- Google **Mock**
  - Matchers
  - **EXPECT_THAT**
  - Mocking Classes and Functions
    - **EXPECT_CALL**
    - **ON_CALL**
  - Advanced Topics
    - Forcing **Sequence**
    - Checkpoints
    - Delegation
    - GMock Custom Matchers

Prerequisite: Installing Bazel

This repo uses `Bazel` for building C++ files. You can install Bazel using this link.

Cloning this repo

```
git clone https://github.com/ourarash/cpp-template.git
```

Examples:

Hello World Example:

You can run this using `bazel`:

```
bazel run src/main:main
```

https://github.com/ourarash/cpp-template

GMock Matchers and EXPECT_THAT

```
TEST(MatchersTest, ExpectEq) {
  EXPECT_EQ(1, 1);
  EXPECT_EQ(2 - 2, 0);
  EXPECT_TRUE(1 == 1);
}
```

With plain GTest macros

```
TEST(MatchersTest, ExpectThat) {
  EXPECT_THAT(1, 1);
  EXPECT_THAT(0, 2 - 2);
  EXPECT_THAT(1 == 1, true);
}
```

Using EXPECT_THAT

```
EXPECT_THAT(actual_value, matcher);
```

Asserts that **actual_value** matches **matcher.**

```
TEST(MatchersTest, SimpleMatcher) {
  testing::Matcher<int> is_one = 1;
  testing::Matcher<int> is_zero = 0;
  testing::Matcher<bool> is_true = true;

  EXPECT_TRUE(is_one.Matches(1));
  EXPECT_TRUE(is_zero.Matches(2 - 2));
  EXPECT_TRUE(is_true.Matches(1 == 1));
}
```

GMock Matchers explicitly defined

```
TEST(MatchersTest, SimpleMatcherExpectThat) {
  testing::Matcher<int> is_one = 1;
  testing::Matcher<int> is_zero = 0;
  testing::Matcher<bool> is_true = true;

  EXPECT_THAT(1, is_one);
  EXPECT_THAT(2 - 2, is_zero);
  EXPECT_THAT(1 == 1, is_true);
}
```

EXPECT_THAT with matchers

tests/gmock_demo/matchers/matchers_intro.c

```
#include "gtest/gtest.h"

TEST(MatchersTest, NoMatcherIsUSed) {

  std::string test_string("Hello, this is a test string. We are here to help!");
  EXPECT_TRUE(test_string.rfind("Hello", 0) == 0);
  EXPECT_TRUE(std::regex_match(test_string, std::regex(".*is.*help.*")));

  int test_value = 10;
  EXPECT_GT(test_value, 0);
  EXPECT_LT(test_value, 100);
}
```

With plain GTest macros

```
#include "gmock/gmock.h"
#include "gtest/gtest.h"

TEST(MatchersTest, MatchersAreUsed) {
  std::string test_string("Hello, this is a test string. We are here to help!");
  EXPECT_THAT(test_string, StartsWith("Hello"));
  EXPECT_THAT(test_string, MatchesRegex(".*is.*help.*"));

  int test_value = 10;
  EXPECT_THAT(test_value, AllOf(Gt(0), Lt(100)));
}
```

With GMock matchers

tests/gmock_demo/matchers/matchers_intro.c

Why Use Matchers?

- **Concise** and **readable**
- Already **tested**
- Can be **chained**
- Can be **customized**

```
TEST(MatchersTest, MatchersAreUsed) {
  std::string test_string(
       "Hello, this is a test string. We are here to help!");
  EXPECT_THAT(test_string, StartsWith("Hello"));
  EXPECT_THAT(test_string, MatchesRegex(".*is.*help.*"));

  int test_value = 10;
  EXPECT_THAT(test_value, AllOf(Gt(0), Lt(100)));
}
```

```
MATCHER_P(IsDivisibleBy, n, "") {
  *result_listener << "where the remainder is "
         << (arg % n);
  return (arg % n) == 0;
}

TEST(MatchersTest, CustomMatchers_ExtraDescription) {
  EXPECT_THAT(9, IsDivisibleBy(2));
  EXPECT_THAT(10, Not(IsDivisibleBy(2)));

  std::vector<int> v = {4, 6, 8, 2};
  EXPECT_THAT(v, Each(IsDivisibleBy(2)));
}
```

A **matcher** matches a *single* argument. You can use it inside `ON_CALL()` or `EXPECT_CALL()`, or use it to validate a value directly using two macros:

Macro	Description
`EXPECT_THAT(actual_value, matcher)`	Asserts that `actual_value` matches `matcher`.
`ASSERT_THAT(actual_value, matcher)`	The same as `EXPECT_THAT(actual_value, matcher)`, except that it generates a **fatal** failure.

```
using ::testing::AllOf;
using ::testing::Gt;
using ::testing::Lt;
using ::testing::MatchesRegex;
using ::testing::StartsWith;
```

```
#include "gmock/gmock.h"
#include "gtest/gtest.h"

TEST(MatchersTest, MatchersAreUsed) {
  std::string test_string("Hello, this is a test string. We are here to help!");
  EXPECT_THAT(test_string, StartsWith("Hello"));
  EXPECT_THAT(test_string, MatchesRegex(".*is.*help.*"));

  int test_value = 10;
  ASSERT_THAT(test_value, AllOf(Gt(0), Lt(100)));
}
```

Comparison Matchers

Generic and Numeric Comparison

```
TEST(GenericComparison, SimpleComparison) {
  int a = 1;
  EXPECT_THAT(a, Eq(1));
  EXPECT_THAT(a, 1);

  EXPECT_THAT(a, Ge(0));
  EXPECT_THAT(a, Gt(0));
  EXPECT_THAT(a, Lt(2));
  EXPECT_THAT(a, Le(2));
  EXPECT_THAT(a, Ne(2));
}
```

tests/gmock_demo/matchers/comparison.cc

Boolean And Pointer Comparison

```
TEST(GenericComparison, BooleanComparison) {
  int boolean_true = true;
  int boolean_false = false;

  EXPECT_THAT(boolean_true, IsTrue());
  EXPECT_THAT(boolean_false, IsFalse());
}
```

```
TEST(GenericComparison, PointerComparison) {
  int* ptr_null = nullptr;
  int* ptr_valid = new int(1);

  EXPECT_THAT(ptr_null, IsNull());
  EXPECT_THAT(ptr_valid, NotNull());
}
```

tests/gmock_demo/matchers/comparison.cc

Reference Matcher

```
TEST(ReferenceTest, SimpleTest) {
  int a = 1;
  int& a_ref = a;
  int b = 1;
  EXPECT_THAT(a_ref, Ref(a));
  EXPECT_THAT(b, Not(Ref(a)));
}
```

tests/gmock_demo/matchers/comparison.cc

TypedEq Matcher

```
TEST(TypedEqTest, SimpleTest) {
  char char_a = 'a';
  int int_a = 'a';
  EXPECT_THAT(char_a, TypedEq<char>('a'));
  EXPECT_THAT(char_a, Not(TypedEq<char>('b')));

  EXPECT_THAT(int_a, TypedEq<int>('a'));
  // Doesn't compile:
  // EXPECT_THAT(int_a, TypedEq<char>('a'));
}
```

tests/gmock_demo/matchers/comparison.cc

Wildcard

```
TEST(Wildcard, Underscore) {
  int a = 1;
  std::string str_a = "aa";

  EXPECT_THAT(a, 1);
  EXPECT_THAT(a, _);
  EXPECT_THAT(str_a, _);
}
```

```
TEST(Wildcard, AnyType) {
  int a = 1;
  auto b = a;

  EXPECT_THAT(a, A<int>());
  EXPECT_THAT(b, An<int>());
  // Doesn't compile:
  // EXPECT_THAT(a, An<char>());
}
```

tests/gmock_demo/matchers/comparison.cc

Optional

```
TEST(OptionalTest, SimpleTest) {
  std::optional<int> opt(1);
  std::optional<int> no_value;
  std::optional<std::string> opt_str = {"AAA"};

  EXPECT_THAT(opt, Optional(1));
  EXPECT_THAT(opt, 1);
  EXPECT_THAT(no_value, Not(Optional(1)));
  EXPECT_THAT(no_value, std::nullopt);

  EXPECT_THAT(opt.has_value(), IsTrue());
  EXPECT_THAT(no_value.has_value(), IsFalse());

  EXPECT_THAT(opt, Optional(Lt(2)));
  EXPECT_THAT(opt_str, (std::string("AAA")));
}
```

tests/gmock_demo/matchers/optional.cc

VariantWith

```
TEST(VariantTest, SimpleTest) {
  std::variant<int, float> var;

  var = 1;
  EXPECT_THAT(var, 1);
  EXPECT_THAT(var, VariantWith<int>(1));
  EXPECT_THAT(std::get<int>(var), 1);
  EXPECT_THAT(var, Not(VariantWith<float>(1)));

  var = 1.1f;
  EXPECT_THAT(var, VariantWith<float>(1.1f));
  EXPECT_THAT(std::get<float>(var), 1.1f);
  EXPECT_THAT(var, 1.1f);
}
```

tests/gmock_demo/matchers/optional.cc

Generic Comparison

Matcher	Description
`Eq(value)` or `value`	`argument == value`
`Ge(value)`	`argument >= value`
`Gt(value)`	`argument > value`
`Le(value)`	`argument <= value`
`Lt(value)`	`argument < value`
`Ne(value)`	`argument != value`
`IsFalse()`	`argument` evaluates to `false` in a Boolean context.
`IsTrue()`	`argument` evaluates to `true` in a Boolean context.
`IsNull()`	`argument` is a `NULL` pointer (raw or smart).
`NotNull()`	`argument` is a non-null pointer (raw or smart).
`Optional(m)`	`argument` is `optional<>` that contains a value matching `m`. (For testing whether an `optional<>` is set, check for equality with `nullopt`. You may need to use `Eq(nullopt)` if the inner type doesn't have `==`.)
`VariantWith<T>(m)`	`argument` is `variant<>` that holds the alternative of type T with a value matching `m`.
`Ref(variable)`	`argument` is a reference to `variable`.
`TypedEq<type>(value)`	`argument` has type `type` and is equal to `value`. You may need to use this instead of `Eq(value)` when the mock function is overloaded.

String Matchers

String Comparison

```
TEST(MatchersTest, StringEquivalence) {
  auto str1 = std::string("This is a test.");
  auto str2 = std::string("THIS IS A TEST.");
  auto str3 = std::string("Another test");

  EXPECT_THAT(str1, StrEq(str1));
  EXPECT_THAT(str1, Eq(str1));
  EXPECT_THAT(str1, str1);

  EXPECT_THAT(str1, StrNe(str2));
  EXPECT_THAT(str1, Ne(str2));
  EXPECT_THAT(str1, Not(Eq(str2)));
  EXPECT_THAT(str1, Not(str2));

  EXPECT_THAT(str1, StrCaseEq(str2));
  EXPECT_THAT(str1, StrCaseNe(str3));
}
```

tests/gmock_demo/matchers/strings.cc

```
TEST(MatchersTest, StringMatchers) {
  auto str1 = std::string("This is aa test. Thanks for watching.");
  auto str2 = std::string("This is bb test.");

  EXPECT_THAT(str1, ContainsRegex(".*aa test\\."));
  EXPECT_THAT(str1, ContainsRegex("[a]+ test\\."));
  EXPECT_THAT(str1, ContainsRegex("a{2} test\\."));
  EXPECT_THAT(str1, ContainsRegex("a{1,} test\\."));
  EXPECT_THAT(str1, ContainsRegex("(aa|bb) test\\."));
  EXPECT_THAT(str2, ContainsRegex("(aa|bb) test\\."));

  EXPECT_THAT(str1, MatchesRegex("T.*\\."));

  EXPECT_THAT(str1, EndsWith("watching."));
  EXPECT_THAT(str1, StartsWith("This is"));

  EXPECT_THAT(str1, HasSubstr("Thanks for"));
}
```

- On **POSIX** systems: POSIX extended regular expression syntax.
- On **Windows**: limited regex syntax. No support for:
  - union ("x|y")
  - grouping ("(xy)")
  - brackets ("[xy]")
  - repetition count ("x{5,7}")

tests/gmock_demo/matchers/strings.cc

String Matchers

String Matchers

The `argument` can be either a C string or a C++ string object:

Matcher	Description
`ContainsRegex(string)`	`argument` matches the given regular expression.
`EndsWith(suffix)`	`argument` ends with string `suffix`.
`HasSubstr(string)`	`argument` contains `string` as a sub-string.
`IsEmpty()`	`argument` is an empty string.
`MatchesRegex(string)`	`argument` matches the given regular expression with the match starting at the first character and ending at the last character.
`StartsWith(prefix)`	`argument` starts with string `prefix`.
`StrCaseEq(string)`	`argument` is equal to `string`, ignoring case.
`StrCaseNe(string)`	`argument` is not equal to `string`, ignoring case.
`StrEq(string)`	`argument` is equal to `string`.
`StrNe(string)`	`argument` is not equal to `string`.

Floating Point Matchers

FloatEq and FloatNear

```
const float a = 67329.234;
const float b = 67329.242; // Exactly 1 ULP away from a
```

```
TEST(FloatingPointTest, ULPComparison){
  EXPECT_NE(a, b);
  EXPECT_FLOAT_EQ(a, b);
  EXPECT_THAT(a, FloatEq(b));
}
```

```
TEST(FloatingPointTest, FloatNear) {
  EXPECT_NEAR(a, b, 0.2);
  EXPECT_THAT(a, FloatNear(b, 0.2));
}
```

ULP (**U**nit in the **L**ast **P**lace or **U**nit of **L**east **P**recision) is the spacing between two consecutive floating-point numbers, i.e., the value the least significant digit represents if it is 1.

- Wikipedia

tests/gmock_demo/matchers/floating_point.cc

IsNan, NanSensitiveFloatEq

```
TEST(FloatingPointTest, NanSensitiveFloatEq) {
  auto qNaN = std::numeric_limits<float>::quiet_NaN();
  auto logNaN = std::log(-1);

  EXPECT_THAT(qNaN, IsNan());
  EXPECT_THAT(logNaN, IsNan());

  EXPECT_THAT(qNaN, NanSensitiveFloatEq(logNaN));
  EXPECT_THAT(qNaN, Not(FloatEq(logNaN)));
}
```

NaN: In IEEE 754, any value with all bits of the exponent set and at least one bit of the fraction set represents a NaN.

`tests/gmock_demo/matchers/floating_point.cc`

Floating Point Matchers

Floating-Point Matchers

Matcher	Description
`DoubleEq(a_double)`	`argument` is a `double` value approximately equal to `a_double`, treating two NaNs as unequal.
`FloatEq(a_float)`	`argument` is a `float` value approximately equal to `a_float`, treating two NaNs as unequal.
`NanSensitiveDoubleEq(a_double)`	`argument` is a `double` value approximately equal to `a_double`, treating two NaNs as equal.
`NanSensitiveFloatEq(a_float)`	`argument` is a `float` value approximately equal to `a_float`, treating two NaNs as equal.
`IsNan()`	`argument` is any floating-point type with a NaN value.

The above matchers use ULP-based comparison (the same as used in googletest). They automatically pick a reasonable error bound based on the absolute value of the expected value. `DoubleEq()` and `FloatEq()` conform to the IEEE standard, which requires comparing two NaNs for equality to return false. The `NanSensitive*` version instead treats two NaNs as equal, which is often what a user wants.

Matcher	Description
`DoubleNear(a_double, max_abs_error)`	`argument` is a `double` value close to `a_double` (absolute error <= `max_abs_error`), treating two NaNs as unequal.
`FloatNear(a_float, max_abs_error)`	`argument` is a `float` value close to `a_float` (absolute error <= `max_abs_error`), treating two NaNs as unequal.
`NanSensitiveDoubleNear(a_double, max_abs_error)`	`argument` is a `double` value close to `a_double` (absolute error <= `max_abs_error`), treating two NaNs as equal.
`NanSensitiveFloatNear(a_float, max_abs_error)`	`argument` is a `float` value close to `a_float` (absolute error <= `max_abs_error`), treating two NaNs as equal.

Composite Matchers

```
TEST(MatchersTest, CompositMatchersSimple) {
  int a = 1;
  EXPECT_THAT(a, AllOf(Eq(1), Gt(0), Lt(5)));
  EXPECT_THAT(a, Not(AllOf(Eq(1), Lt(0), Gt(5))));
  EXPECT_THAT(a, AnyOf(Eq(1), Lt(0), Gt(5)));
}
```

```
TEST(MatchersTest, CompositMatchers) {
  auto str = std::string("This is aa test. Thanks for watching.");
  EXPECT_THAT(str, AllOf(HasSubstr("aa"), HasSubstr("test")));
  EXPECT_THAT(str, Not(AllOf(HasSubstr("aa"), HasSubstr("bb"))));
  EXPECT_THAT(str, AnyOf(HasSubstr("aa"), HasSubstr("bb")));

  std::vector<Matcher<std::string>> matchers = {EndsWith("watching."),
  HasSubstr("Thanks for")};
  EXPECT_THAT(str, AllOfArray(matchers));

  bool selector = true;
  EXPECT_THAT(str, Conditional(selector, HasSubstr("aa"), HasSubstr("bb")));
}
```

tests/gmock_demo/matchers/composite.cc

Matcher	Description
`AllOf(m1, m2, ..., mn)`	`argument` matches all of the matchers `m1` to `mn`.
`AllOfArray({m0, m1, ..., mn})`, `AllOfArray(a_container)`, `AllOfArray(begin, end)`, `AllOfArray(array)`, or `AllOfArray(array, count)`	The same as `AllOf()` except that the matchers come from an initializer list, STL-style container, iterator range, or C-style array.
`AnyOf(m1, m2, ..., mn)`	`argument` matches at least one of the matchers `m1` to `mn`.
`AnyOfArray({m0, m1, ..., mn})`, `AnyOfArray(a_container)`, `AnyOfArray(begin, end)`, `AnyOfArray(array)`, or `AnyOfArray(array, count)`	The same as `AnyOf()` except that the matchers come from an initializer list, STL-style container, iterator range, or C-style array.
`Not(m)`	`argument` doesn't match matcher `m`.
`Conditional(cond, m1, m2)`	Matches matcher `m1` if `cond` evaluates to true, else matches `m2`.

tests/gmock_demo/matchers/composite.cc

Container Matchers

ElementsAre, UnorderedElementsAre

```
TEST(MatchersTest, ElementsAre) {
  std::vector<int> v = {3, 4, 5, 2, 6, 7, 8, 9, 10};

  EXPECT_THAT(v, ElementsAre(3, 4, 5, 2, 6, 7, 8, 9, 10));
}
```

```
TEST(MatchersTest, UnorderedElementsAre) {
  std::vector<int> v = {3, 4, 5, 2};
  EXPECT_THAT(v, UnorderedElementsAre(4, 3, 5, 2));
}
```

tests/gmock_demo/matchers/container.cc

WhenSorted

```
TEST(MatchersTest, WhenSorted) {
  std::vector<int> v = {3, 4, 5, 2, 6, 7, 8, 9, 10};

  EXPECT_THAT(v, WhenSorted(ElementsAre(2, 3, 4, 5, 6, 7, 8, 9, 10)));
  EXPECT_THAT(v, WhenSortedBy(std::greater<int>(),
                 ElementsAre(10, 9, 8, 7, 6, 5, 4, 3, 2)));
}
```

tests/gmock_demo/matchers/container.cc

(Unordered)ElementsAreArray

```
TEST(MatchersTest, UnorderedElementsAreArray) {
  std::vector<int> v = {3, 4, 5, 2, 6, 7, 8, 9, 10};
  EXPECT_THAT(v, UnorderedElementsAreArray({2, 3, 4, 5, 6, 7, 8, 9, 10}));
  EXPECT_THAT(v, UnorderedElementsAreArray(v.begin(), v.end()));
}
```

```
TEST(MatchersTest, UnorderedElementsAreArray_Set) {
  std::vector<int> v = {3, 4, 5, 2, 6, 7, 8, 9, 10};
  std::set<int> s(v.begin(), v.end());

  EXPECT_THAT(v, UnorderedElementsAreArray(s.begin(), s.end()));
}
```

tests/gmock_demo/matchers/container.cc

Each, Contains, Subset

```
TEST(MatchersTest, ContainerMatchers) {
  std::vector<int> v = {4, 6, 8, 2, 2};
  std::vector<int> u = {4, 6};

  EXPECT_THAT(v, Each(Gt(1)));
  EXPECT_THAT(v, Contains(6));
  EXPECT_THAT(u, IsSubsetOf(v));

  // Contains an item that is greater than 1.
  EXPECT_THAT(v, Contains(Gt(1)));

  // Each element is either greater than 2 or greater than 0
  EXPECT_THAT(v, Each(AnyOf(Gt(2), Gt(0))));
}
```

tests/gmock_demo/matchers/container.cc

Contains Times

```
TEST(MatchersTest, ContainerMatchers_ContainsNTimes) {
  std::vector<int> v = {4, 6, 8, 2, 2, 3, 3, 3, 4};

  EXPECT_THAT(v, Contains(3).Times(3));
  EXPECT_THAT(v, Contains(4).Times(2));
  EXPECT_THAT(v, Contains(1).Times(0));
}
```

tests/gmock_demo/matchers/container.cc

Pointwise

```
TEST(MatchersTest, ContainerMatchers_Pointwise) {
  std::list<int> lhs{1, 2, 3};
  std::vector<int> rhs{4, 5, 6};

  EXPECT_THAT(lhs, Pointwise(Lt(), rhs));
  EXPECT_THAT(lhs, Not(Pointwise(Gt(), rhs)));
}
```

tests/gmock_demo/matchers/container.cc

ContainerEq

```
TEST(MatchersTest, ContainerMatchers_ContainerEq) {
  std::vector<int> v1{1, 2, 3, 4, 5};
  std::vector<int> v2{1, 2, 3};

  EXPECT_THAT(v1, v2);
  EXPECT_THAT(v1, ContainerEq(v2));
}
```

```
[ RUN      ] MatchersTest.ContainerMatchers_ContainerEq
tests/gmock_demo/matchers/container.cc:107: Failure
Value of: v1
Expected: is equal to { 1, 2, 3 }
  Actual: { 1, 2, 3, 4, 5 }
tests/gmock_demo/matchers/container.cc:108: Failure
Value of: v1
Expected: equals { 1, 2, 3 }
  Actual: { 1, 2, 3, 4, 5 }, which has these unexpected elements: 4, 5
[  FAILED  ] MatchersTest.ContainerMatchers_ContainerEq (0 ms)
```

tests/gmock_demo/matchers/container.cc

Container Matchers

Matcher	Description
`BeginEndDistanceIs(m)`	`argument` is a container whose `begin()` and `end()` iterators are separated by a number of increments matching `m`. E.g. `BeginEndDistanceIs(2)` or `BeginEndDistanceIs(Lt(2))`. For containers that define a `size()` method, `SizeIs(m)` may be more efficient.
`ContainerEq(container)`	The same as `Eq(container)` except that the failure message also includes which elements are in one container but not the other.
`Contains(e)`	`argument` contains an element that matches `e`, which can be either a value or a matcher.
`Contains(e).Times(n)`	`argument` contains elements that match `e`, which can be either a value or a matcher, and the number of matches is `n`, which can be either a value or a matcher. Unlike the plain `Contains` and `Each` this allows to check for arbitrary occurrences including testing for absence with `Contains(e).Times(0)`.
`Each(e)`	`argument` is a container where *every* element matches `e`, which can be either a value or a matcher.
`ElementsAre(e0, e1, ..., en)`	`argument` has `n + 1` elements, where the *i*-th element matches `ei`, which can be a value or a matcher.
`ElementsAreArray({e0, e1, ..., en})`, `ElementsAreArray(a_container)`, `ElementsAreArray(begin, end)`, `ElementsAreArray(array)`, or `ElementsAreArray(array, count)`	The same as `ElementsAre()` except that the expected element values/matchers come from an initializer list, STL-style container, iterator range, or C-style array.
`IsEmpty()`	`argument` is an empty container (`container.empty()`).
`IsSubsetOf({e0, e1, ..., en})`, `IsSubsetOf(a_container)`, `IsSubsetOf(begin, end)`, `IsSubsetOf(array)`, or `IsSubsetOf(array, count)`	`argument` matches `UnorderedElementsAre(x0, x1, ..., xk)` for some subset `{x0, x1, ..., xk}` of the expected matchers.
`IsSupersetOf({e0, e1, ..., en})`, `IsSupersetOf(a_container)`, `IsSupersetOf(begin, end)`, `IsSupersetOf(array)`, or `IsSupersetOf(array, count)`	Some subset of `argument` matches `UnorderedElementsAre(` expected matchers `)`.
`Pointwise(m, container)`, `Pointwise(m, {e0, e1, ..., en})`	`argument` contains the same number of elements as in `container`, and for all i, (the i-th element in `argument`, the i-th element in `container`) match `m`, which is a matcher on 2-tuples. E.g. `Pointwise(Le(), upper_bounds)` verifies that each element in `argument` doesn't exceed the corresponding element in `upper_bounds`. See more detail below.
`SizeIs(m)`	`argument` is a container whose size matches `m`. E.g. `SizeIs(2)` or `SizeIs(Lt(2))`.
`UnorderedElementsAre(e0, e1, ..., en)`	`argument` has `n + 1` elements, and under *some* permutation of the elements, each element matches an `ei` (for a different `i`), which can be a value or a matcher.
`UnorderedElementsAreArray({e0, e1, ..., en})`, `UnorderedElementsAreArray(a_container)`, `UnorderedElementsAreArray(begin, end)`, `UnorderedElementsAreArray(array)`, or `UnorderedElementsAreArray(array, count)`	The same as `UnorderedElementsAre()` except that the expected element values/matchers come from an initializer list, STL-style container, iterator range, or C-style array.
`UnorderedPointwise(m, container)`, `UnorderedPointwise(m, {e0, e1, ..., en})`	Like `Pointwise(m, container)`, but ignores the order of elements.
`WhenSorted(m)`	When `argument` is sorted using the `<` operator, it matches container matcher `m`. E.g. `WhenSorted(ElementsAre(1, 2, 3))` verifies that `argument` contains elements 1, 2, and 3, ignoring order.
`WhenSortedBy(comparator, m)`	The same as `WhenSorted(m)`, except that the given comparator instead of `<` is used to sort `argument`. E.g. `WhenSortedBy(std::greater(), ElementsAre(3, 2, 1))`.

Member Matchers

Pair, Key, FieldsAre

```
TEST(MatchersTest, MemberMatchers) {
  std::map<int, int> m = {{1, 100}, {2, 200}, {3, 300}};
  // Each key is greater than 0 and each value is greater than 99.
  EXPECT_THAT(m, Each(Pair(Gt(0), Gt(99))));
  // Each key is greater than 0.
  EXPECT_THAT(m, Each(Key(Gt(0))));
  // The value is greater than 99.
  EXPECT_THAT(m, Each(Pair(_, Gt(99))));
  // The first part of each pair is greater than 0 and the second part is
  // greater than 100 or equal to 100.
  EXPECT_THAT(m, Each(FieldsAre(Gt(0), AnyOf(Gt(100), Eq(100)))));
}
```

tests/gmock_demo/matchers/member.cc

FieldsAre for Tuple and Struct

```
TEST(MatchersTest, MemberMatchers_Field_Tuple) {
  std::tuple<int, std::string> my_tuple{7, "hello world"};
  EXPECT_THAT(my_tuple, FieldsAre(Ge(0), HasSubstr("hello")));
}
```

```
// Requires C++17
TEST(MatchersTest, MemberMatchers_Field_Struct) {
  struct MyStruct {
    int value = 42;
    std::string greeting = "aloha";
  };
  MyStruct s;
  EXPECT_THAT(s, FieldsAre(42, "aloha"));
}
```

tests/gmock_demo/matchers/member.cc

```
class MyClass1 {
 public:
  MyClass1(int m) : m_(m) {}

  int m_;
};
```

```
Field(&MyClass1::Field, m)
```

Creates a **matcher** that matches an object whose given **Field** matches **m.**

```
TEST(MatchersTest, MemberMatchers_Field_Simple) {
  MyClass1 my_object(1);
  EXPECT_THAT(my_object, Field(&MyClass1::m_, Ge(0)));
  EXPECT_THAT(my_object.m_, Ge(0));
}
```

```
TEST(MatchersTest, MemberMatchers_Field_Container) {
  std::vector<MyClass1> my_objects = {MyClass1(1), MyClass1(2), MyClass1(3)};

  EXPECT_THAT(my_objects, Each(Field(&MyClass1::m_, Ge(0))));
  EXPECT_THAT(my_objects, UnorderedElementsAre(Field(&MyClass1::m_, 1),
                                               Field(&MyClass1::m_, 3),
                                               Field(&MyClass1::m_, 2)));

}
```

tests/gmock_demo/matchers/member.

```
class MyClass2 {
  public:
    MyClass2(int m) : m_(m) {}
    int GetM() const { return m_; }
  private:
    int m_;
};
```

```
Property(&MyClass2::Property, m)
```

Creates a **matcher** that matches an object whose given **Property** matches **m**. The method **Property()** must take no argument and be declared as **const**.

```
TEST(MatchersTest, MemberMatchers_Property_simple) {
  MyClass2 my_object(1);
  EXPECT_THAT(my_object, Property(&MyClass2::GetM, Ge(0)));

  EXPECT_THAT(my_object.GetM(), Ge(0));
}
```

```
TEST(MatchersTest, MemberMatchers_Property_Container) {
  std::vector<MyClass2> my_objects = {MyClass2(1), MyClass2(2), MyClass2(3)};

  EXPECT_THAT(my_objects, Each(Property(&MyClass2::GetM, Ge(0))));
  EXPECT_THAT(my_objects, UnorderedElementsAre(Property(&MyClass2::GetM, 1),
                                               Property(&MyClass2::GetM, 3),
                                               Property(&MyClass2::GetM, 2)));
}
```

tests/gmock_demo/matchers/member.

Member Matchers

Member Matchers

Matcher	Description
`Field(&class::field, m)`	`argument.field` (or `argument->field` when `argument` is a plain pointer) matches matcher `m`, where `argument` is an object of type *class*.
`Field(field_name, &class::field, m)`	The same as the two-parameter version, but provides a better error message.
`Key(e)`	`argument.first` matches `e`, which can be either a value or a matcher. E.g. `Contains(Key(Le(5)))` can verify that a `map` contains a key `<= 5`.
`Pair(m1, m2)`	`argument` is an `std::pair` whose `first` field matches `m1` and `second` field matches `m2`.
`FieldsAre(m...)`	`argument` is a compatible object where each field matches piecewise with the matchers `m...`. A compatible object is any that supports the `std::tuple_size<Obj>` + `get<I>(obj)` protocol. In C++17 and up this also supports types compatible with structured bindings, like aggregates.
`Property(&class::property, m)`	`argument.property()` (or `argument->property()` when `argument` is a plain pointer) matches matcher `m`, where `argument` is an object of type *class*. The method `property()` must take no argument and be declared as `const`.
`Property(property_name, &class::property, m)`	The same as the two-parameter version, but provides a better error message.

Pointer Matchers

```
void Test() {
  // Pointee of a_ptr is a;
  // Pointer of a_ptr is a_ptr;
  // Address of a is in a_ptr.
  int a = 1;
  int *a_ptr = &a;
}
```

```
void Test() {
  // Pointee of u_ptr is 10.
  // pointer of u_ptr is u_ptr.get().
  // Address of 10 is in u_ptr.
  std::unique_ptr<int> u_ptr(new int(10));
  int *pointer = u_ptr.get();
}
```

Pointer Matchers

Matcher	Description
`Address(m)`	the result of `std::addressof(argument)` matches `m`.
`Pointee(m)`	`argument` (either a smart pointer or a raw pointer) points to a value that matches matcher `m`.
`Pointer(m)`	`argument` (either a smart pointer or a raw pointer) contains a pointer that matches `m`. `m` will match against the raw pointer regardless of the type of `argument`.
`WhenDynamicCastTo<T>(m)`	when `argument` is passed through `dynamic_cast<T>()`, it matches matcher `m`.

`EXPECT_THAT(a, Pointee(m)):` **a** points to a value that matches **m.**
OR:
When **a** is **dereferenced**, it matches **m**.

```
TEST(PointerTest, Pointee_RawPointer_Simple) {
  int *n_ptr = new int(10);
  EXPECT_THAT(*n_ptr, Ge(9));
  EXPECT_THAT(n_ptr, Pointee(Ge(9)));
}
```

```
TEST(PointerTest, Pointee_RawPointer_Container) {
  int a = 1, b = 2;
  std::vector<int *> my_pointers = {&a, &b};

  EXPECT_THAT(my_pointers, Each(Pointee(Ge(0))));
  EXPECT_THAT(my_pointers, UnorderedElementsAre(Pointee(b), Pointee(a)));

  // Doesn't work!
  // EXPECT_THAT(my_pointers, UnorderedElementsAre(b, a));
}
```

tests/gmock_demo/matchers/pointer.cc

`EXPECT_THAT(a, Pointee(m)):` **a** points to a value that matches **m.**
OR:
When **a** is **dereferenced**, it matches **m**.

```
TEST(PointerTest, Pointee_SmartPointer_Simple) {
  std::unique_ptr<int> n_ptr(new int(10));
  EXPECT_THAT(*n_ptr, Ge(9));
  EXPECT_THAT(n_ptr, Pointee(Ge(0)));
}
```

```
TEST(PointerTest, Pointee_SmartPointer_Container) {
  std::vector<std::unique_ptr<int>> my_pointers;
  my_pointers.push_back(std::unique_ptr<int>(new int(1)));
  my_pointers.push_back(std::unique_ptr<int>(new int(2)));

  EXPECT_THAT(my_pointers, Each(Pointee(Ge(0))));
  EXPECT_THAT(my_pointers, UnorderedElementsAre(Pointee(2), Pointee(1)));
}
```

`tests/gmock_demo/matchers/pointer.cc`

EXPECT_THAT(a, Address(m)): the result of **std::addressof(a)** matches **m**.

```
TEST(PointerTest, Address) {
  int n = 10;
  int *n_ptr1 = &n, *n_ptr2 = &n;

  EXPECT_THAT(n, Address(n_ptr1));
  EXPECT_THAT(n, Address(AllOf(n_ptr1, n_ptr2)));
}
```

```
TEST(PointerTest, Address_Container2) {
  std::vector<int> my_objects = {1, 2};

  EXPECT_THAT(my_objects, UnorderedElementsAre(Address(&my_objects[0] + 1),
                                               Address(&my_objects[0])));
}
```

tests/gmock_demo/matchers/pointer.cc

EXPECT_THAT(a, Pointer(m)): **a** contains a pointer that matches **m**.

```
TEST(PointerTest, Pointer_RawPointer) {
  int n = 10;
  int *n_ptr1 = &n, *n_ptr2 = &n;
  EXPECT_THAT(&n, Pointer(&n));
  EXPECT_THAT(n_ptr1, Pointer(&n));
  EXPECT_THAT(&n, AllOf(Pointer(n_ptr1), Pointer(n_ptr2)));
}
```

tests/gmock_demo/matchers/pointer.cc

EXPECT_THAT(a, Pointer(m)): **a** contains a pointer that matches **m**.

```
TEST(PointerTest, Pointer_SmartPointer_Simple) {
  std::unique_ptr<int> n(new int(10));
  EXPECT_THAT(n, Pointer(n.get()));
}
```

```
TEST(PointerTest, Pointer_SmartPointer_Simple2) {
  std::unique_ptr<int> a(new int(1));
  std::unique_ptr<int> b(new int(1));
  EXPECT_THAT(a, Pointer(a.get()));
  EXPECT_THAT(a, Not(Pointer(b.get())));
}
```

tests/gmock_demo/matchers/pointer.cc

EXPECT_THAT(a, Pointer(m)): **a** contains a pointer that matches **m**.

```
TEST(PointerTest, Pointer_SmartPointer_Container) {
  std::vector<std::unique_ptr<int>> my_pointers;
  my_pointers.push_back(std::unique_ptr<int>(new int(1)));
  my_pointers.push_back(std::unique_ptr<int>(new int(2)));

  EXPECT_THAT(my_pointers, UnorderedElementsAre(
    Pointer(my_pointers[1].get()),
    Pointer(my_pointers[0].get())));
}
```

tests/gmock_demo/matchers/pointer.cc

EXPECT_THAT(a, Pointer(m)): **a** contains a pointer that matches **m**.

```
TEST(PointerTest, Pointer_SmartPointer_Container) {
  std::vector<std::unique_ptr<int>> my_pointers;
  my_pointers.push_back(std::unique_ptr<int>(new int(1)));
  my_pointers.push_back(std::unique_ptr<int>(new int(2)));

  EXPECT_THAT(my_pointers, UnorderedElementsAre(
    Pointer(my_pointers[1].get()),
    Pointer(my_pointers[0].get())));
}
```

```
TEST(PointerTest, Pointer_SharedPointer) {
  auto a_ptr = std::make_shared<int>(1);
  auto b_ptr = a_ptr;
  std::vector<std::shared_ptr<int>> my_pointers = {a_ptr, b_ptr};

  EXPECT_THAT(my_pointers, UnorderedElementsAre(a_ptr, b_ptr));
}
```

tests/gmock_demo/matchers/pointer.cc

EXPECT_THAT(a, WhenDynamicCastTo<T>(m)): when **a** is passed through dynamic_cast<T>(), it matches matcher **m**.

```
class Base {
 public:
  Base() {}
  virtual ~Base() {}
};
```

```
class Derived : public Base {
 public:
  Derived() { }
  int i;
};
```

```
TEST(WhenDynamicCastToTest, SimpleTest) {
  Derived derived;
  derived.i = 4;

  Base *base_ptr_derived = &derived;
  Base *base_ptr = new Base();

  EXPECT_THAT(dynamic_cast<Derived *>(base_ptr_derived), Not(IsNull()));
  EXPECT_THAT(dynamic_cast<Derived *>(base_ptr_derived), Field(&Derived::i, 4));
  EXPECT_THAT(dynamic_cast<Derived *>(base_ptr), IsNull());

  EXPECT_THAT(base_ptr_derived, WhenDynamicCastTo<Derived *>(Not(IsNull())));
  EXPECT_THAT(base_ptr_derived, WhenDynamicCastTo<Derived *>(Field(&Derived::i, 4)));
}
```

tests/gmock_demo/matchers/pointer.cc

Predicates

GMock Truly

```
// Returns non-zero if the input is positive.
bool IsPositive(int x) { return x > 0; }
```

```
TEST(ExpectPredTest, IsPositive) {
  EXPECT_TRUE(IsPositive(10));
  EXPECT_PRED1(IsPositive, 10);
}
```

```
TEST(ExpectPredTest, IsPositiveTruly) {
  EXPECT_THAT(10, Truly(IsPositive));
}
```

With GMock Truly

```
TEST(MatchersTest, WithoutTruly) {
  std::vector<double> v = {4, 6, 8, 2};

  for (auto& e : v) {
    EXPECT_TRUE(IsPositive(e));
    EXPECT_PRED1(IsPositive, e);
  }
}
```

Without GMock Truly

```
TEST(MatchersTest, WithTruly) {
  std::vector<double> v = {4, 6, 8, 2};

  EXPECT_THAT(v, Each(Truly(IsPositive)));
}
```

With GMock Truly

tests/gmock_demo/matchers/predicates.cc

Custom Matchers

```
TEST(MatchersTest, CustomMatchers) {
  EXPECT_THAT(9, IsEven());
  EXPECT_THAT(10, Not(IsEven()));
  EXPECT_THAT(100, IsBetween(9, 11));
}
```

GMock Custom Matchers

```
MATCHER(name, description_string_expression) { statements; }
```

- It must return **bool.**
- The value being matched can be referred to by **`arg`** and its type can be referred to by **`arg_type`**.
- The ***description string***: generates the **failure message**.
  - It can (and should) reference the special bool variable **`negation`**.
- **Optionally**, you can stream **additional information** to a hidden argument named **`result_listener`** to explain the match result.

```
MATCHER(IsEven, "") {
  return (arg % 2) == 0;
}
```

```
TEST(MatchersTest, CustomMatchers) {
  EXPECT_THAT(9, IsEven());
  EXPECT_THAT(10, Not(IsEven()));
}
```

```
tests/gmock_demo/matchers/matcher_examples.cc:114:
Failure
Value of: 9
Expected: is even
Actual: 9 (of type int)

tests/gmock_demo/matchers/matcher_examples.cc:115:
Failure
Value of: 10
Expected: not (is even)
Actual: 10 (of type int)
```

GMock Custom Parameterized Matchers

```
MATCHER_Pk(name, param_name, description_string) { statements; }
```

```
MATCHER_P2(IsBetween, a, b,
    (std::string(negation ? "Isn't" : "Is")
    + " between " + PrintToString(a)
    + " and " + PrintToString(b))) {
  return a <= arg && arg <= b;
}
```

```
TEST(CustomeMatchersTest, WithDescription) {
  EXPECT_THAT(100, IsBetween(9, 11));
  EXPECT_THAT(10, Not(IsBetween(9, 11)));
}
```

```
tests/gmock_demo/matchers/matcher_examples.cc:119: Failure
Value of: 100
Expected: Is between 9 and 11
Actual: 100 (of type int)

tests/gmock_demo/matchers/matcher_examples.cc:120: Failure
Value of: 10
Expected: Isn't between 9 and 11
Actual: 10 (of type int)
```

- It must return **bool.**
- The value being matched can be referred to by **`arg`** and its type can be referred to by **`arg_type`**.
- The ***description string***: generates the **failure message**.
  - It can (and should) reference the special bool variable **`negation`**.
- **Optionally**, you can stream **additional information** to a hidden argument named **`result_listener`** to explain the match result.

`tests/gmock_demo/matchers/custom_matchers.cc`

GMock Custom Matchers with More Description

```
MATCHER_P(IsDivisibleBy, n, "") {
  *result_listener << "where the remainder is "
       << (arg % n);
  return (arg % n) == 0;
}
TEST(CustomeMatchersTest, ExtraDescription) {
  EXPECT_THAT(9, IsDivisibleBy(2));
  EXPECT_THAT(10, Not(IsDivisibleBy(2)));

  std::vector<int> v = {4, 6, 8, 2};
  EXPECT_THAT(v, Each(IsDivisibleBy(2)));
}
```

```
tests/gmock_demo/matchers/matcher_examples.cc:124: Failure
Value of: 9
Expected: is divisible by 2
Actual: 9 (of type int), where the remainder is 1

tests/gmock_demo/matchers/matcher_examples.cc:125: Failure
Value of: 10
Expected: not (is divisible by 2)
Actual: 10 (of type int), where the remainder is 0
```

- It must return **bool.**
- The value being matched can be referred to by **`arg`** and its type can be referred to by **`arg_type`**.
- The ***description string***: generates the **failure message**.
  - It can (and should) reference the special bool variable **`negation`**.
- **Optionally**, you can stream **additional information** to a hidden argument named **`result_listener`** to explain the match result.

`tests/gmock_demo/matchers/custom_matchers.cc`

Custom Matchers Composite

```
TEST(MatchersTest, CustomMatchers_Composite) {
  std::vector<int> v = {4, 6, 8, 2};

  EXPECT_THAT(v, Each(IsDivisibleBy(2)));

  EXPECT_THAT(v,
              ElementsAre(Lt(5), IsEven(), IsBetween(1, 10), Not(IsBetween(5, 10))));

  EXPECT_THAT(v, Each(Not(IsBetween(-1, -2))));
}
```

tests/gmock_demo/matchers/custom_matchers.cc

```
MATCHER(IsEven, "") { return (arg % 2) == 0; }
MATCHER(IsOdd, "") { return (arg % 2) != 0; }

MATCHER_P(IsDivisibleBy, n, "") {
  *result_listener << "where the remainder is " << (arg % n);
  return (arg % n) == 0;
}

MATCHER_P2(IsBetween, a, b,
  (std::string(negation ? "Isn't" : "Is") + " between " +
  PrintToString(a) + " and " + PrintToString(b))) {
  return a <= arg && arg <= b;
}

MATCHER_P3(IsOneOf, a, b, c, "") {
  return arg == a || arg == b || arg == c;
}
```

```
tests/gmock_demo/matchers/matcher_examples.cc:124: Failure
Value of: 9
Expected: is divisible by 2
Actual: 9 (of type int), where the remainder is 1

tests/gmock_demo/matchers/matcher_examples.cc:125: Failure
Value of: 10
Expected: not (is divisible by 2)
Actual: 10 (of type int), where the remainder is 0
```

```
tests/gmock_demo/matchers/matcher_examples.cc:114:
Failure
Value of: 9
Expected: is even
Actual: 9 (of type int)

tests/gmock_demo/matchers/matcher_examples.cc:115:
Failure
Value of: 10
Expected: not (is even)
Actual: 10 (of type int)
```

What is Next?

- Google **Test**
  - Google Test Macros
  - Test **Fixtures**
  - Manipulating Tests
  - Advanced Topics
    - Sharing Test Logic
    - Typed Test
    - Type-Parameterized Test
    - Assertion in Subroutines
    - Testing Private Code

- Google **Mock**
  - Matchers
  - **EXPECT_THAT**
  - Mocking Classes and Functions
    - **EXPECT_CALL**
    - **ON_CALL**
  - Advanced Topics
    - Forcing **Sequence**
    - Checkpoints
    - Delegation
    - GMock Custom Matchers