igloo - BDD Style Unit Testing for C++

Assertions

Igloo uses a constraint based assertion model that is heavily inspired by the model used in NUnit. An assertion in Igloo is written using the following format:

```
Assert::That(actual_value, <constraint expression>);
```

where <constraint expression> is an expression that actual_value is evaluated against when the test is executed.

Constraint expressions come in two basic forms: composite and fluent expressions

Composite Expressions

With composite expressions, you can create compact, powerful expressions that combine a set of predefined constraints with ones that you provide yourself.

Example:

```
Assert::That(length, IsGreaterThan(4) && !Equals(10));
```

Composite expressions can be any combination of constraints and the standard logical C++ operators.

You can also add your own constraints to be used within composite expressions.

Fluent Expressions

With fluent expressions, you can create assertions that better convey the intent of a test without exposing implementation-specific details. Fluent expressions aim to help you create tests that are not just by developers for developers, but rather can be read and understood by domain experts.

Fluent expressions also has the ability to make assertions on the elements in a conteiner in a way you cannot achieve with composite expressions.

Example:

```
Assert::That(length, Is().GreaterThan(4).And().Not().EqualTo(10));
```

Basic Constraints

Equality Constraint

Used to verify equality between actual and expected.

```
Assert::That(x, Equals(12)); Assert::That(x, Is().EqualTo(12));
```

EqualityWithDelta Constraint

Used to verify equality between actual and expected, allowing the two to differ by a delta.

```
Assert::That(2.49, EqualsWithDelta(2.5, 0.1)); Assert::That(2.49, Is().EqualToW
ithDelta(2.5, 0.1));
```

GreaterThan Constraint

Used to verify that actual is greater than a value.

```
Assert::That(x, IsGreaterThan(4)); Assert::That(x, Is().GreaterThan(4));
```

LessThan Constraint

Used to verify that actual is less than a value.

```
Assert::That(x, IsLessThan(3)); Assert::That(x, Is().LessThan(3));
```

String Constraints

String assertions in Igloo are used to verify the values of STL strings (std::string).

Equality Constraints

Used to verify that actual is equal to an expected value.

```
Assert:That(actual_str, Equals("foo")); Assert::That(actual_str, Is().EqualTo("
foo"));
```

Contains Constraint

Used to verify that a string contains a substring.

```
Assert::That(actual_str, Contains("foo")); Assert::That(actual_str, Is().Contai
ning("foo"));
```

EndsWith Constraint

Used to verify that a string ends with an expected substring.

```
Assert::That(actual_str, EndsWith("foo")); Assert::That(actual_str, Is().Ending
With("foo"));
```

StartsWith Constraint

Used to verify that a string starts with an expected substring.

```
Assert::That(actual_str, StartsWith("foo")); Assert::That(actual_str, Is().Starting
tingWith("foo"));
```

HasLength Constraint

Used to verify that a string is of the expected length.

```
Assert::That(actual_str, HasLength(5)); Assert::That(actual_str, Is().OfLength
(5));
```

Constraints on Multi Line Strings

If you have a string that contains multiple lines, you can use the collection constraints to make assertions on the content of that string. This may be handy if you have a string that, for instance, represents the resulting content of a file or a network transmission.

Igloo can handle both windows (CR+LF) and unix (LF) line endings

```
std::string lines = "First line\r\nSecond line\r\nThird line"; Assert::That(lin
es, Has().Exactly(1).StartingWith("Second"));
```

Container Constraints

The following constraints can be applied to containers in the standard template library:

Contains Constraint

Used to verify that a container contains an expected value.

```
Assert::That(container, Contains(12)); Assert::That(container, Is().Containing
(12));
```

HasLength Constraint

Used to verify that a container has the expected length.

```
Assert::That(container, HasLength(3)); Assert::That(container, Is().OfLength
(3));
```

IsEmpty Constraint

Used to verify that a container is empty.

```
Assert::That(contatiner, IsEmpty()); Assert::That(container, Is().Empty());
```

All

Used to verify that all elements of a STL sequence container matches an expectation.

```
Assert::That(container, Has().All().LessThan(5).Or().EqualTo(66));
```

AtLeast

Used to verify that at least a specified amount of elements in a STL sequence container matches an expectation.

```
Assert::That(container, Has().AtLeast(3).StartingWith("foo"));
```

AtMost

Used to verify that at most a specified amount of elements in a STL sequence container matches an expectation.

```
Assert:That(container, Has().AtMost(2).Not().Containing("failed"));
```

Exactly

Used to verify that a STL sequence container has exactly a specified amount of elements that matches an expectation.

```
Assert::That(container, Has().Exactly(3).GreaterThan(10).And().LessThan(20));
```

EqualsContainer

Used to verify that two STL sequence containers are equal.

```
Assert::That(container1, EqualsContainer(container2)); Assert::That(container1,
 Is().EqualToContainer(container2));
```

Predicate functions

You can supply a predicate function or a functor to EqualsContainer to customize how to compare the elements in the two containers.

With a predicate function:

```
static bool are_my_types_equal(const my_type& lhs, const my_type& rhs) { return
 lhs.my_val_ == rhs.my_val_; } Assert::That(container1, EqualsContainer(contain
er2, are_my_types_equal));
```

With a functor as predicate:

```
struct within_delta { within_delta(int delta) : delta_(delta) {} bool operator
()(const my_type& lhs, const my_type& rhs) const { return abs(lhs.my_val_ - rh
s.my_val_) <= delta_; } private: int delta_; }; Assert::That(container1, Is().E
qualToContainer(container1, within_delta(1));
```

Exceptions

Exception constraints can be used to verify that your code throws the correct exceptions.

AssertThrows

AssertThrows succeeds if the exception thrown by the call is of the supplied type (or one of its subtypes).

```
AssertThrows(std::logic_error, myObject.a_method(42));
```

Making Assertions on the Thrown Exceptions

If AssertThrows succeeds, it will store the thrown exception so that you can make more detailed assertions on it.

```
AssertThrows(std::logic_error, myObject.a_method(42)); Assert::That(LastExcepti
on<std::logic_error>().what(), Is().Containing("logic failure"));
```

The LastException<> is available in the scope of the call to AssertThrows. An exception is not available between specs in order to avoid the result of one spec contaminating another.

Custom Constraints

You can add your own constraints to Igloo to create more expressive specifications.

Fulfills Constraints

By defining the following matcher

```
1 struct IsEvenNumber 2 { 3 bool Matches(const int actual) const 4 { 5 return
(actual % 2) == 0; 6 } 7 8 friend std::ostream& operator<<(std::ostream& stm, c
onst IsEvenNumber& ); 9 }; 10 11 std::ostream& operator<<(std::ostream& stm, co
nst IsEvenNumber& ) 12 { 13 stm << "An even number"; 14 return stm; 15 }
```

You can create the following constraints in Igloo:

```
Assert::That(42, Fulfills(IsEvenNumber())); Assert::That(42, Is().Fulfilling(Is
EvenNumber()));
```

Your custom matcher should implement a method called Matches() that takes a parameter of the type you expect and returns true if the passed parameter fulfills the constraint.

To get more expressive failure messages, you should also implement the streaming operator as in the example above.