

Testing

# OBJECT-ORIENTED CODE

---

## Objectives

At the end of this section, you will be able to

- Explain the difference between testing C++ constructors and methods
- Create and execute tests for C++ methods

# testIDEA

## Contents

# OBJECT-ORIENTED CODE

---

|   |                               |       |
|---|-------------------------------|-------|
| 1 | Basics of testing C++ classes | 3-5   |
| 2 | Testing C++ constructors      | 6-14  |
| 3 | Testing a C++ method          | 15-23 |
| 4 | Summary                       | 24-25 |

# testIDEA

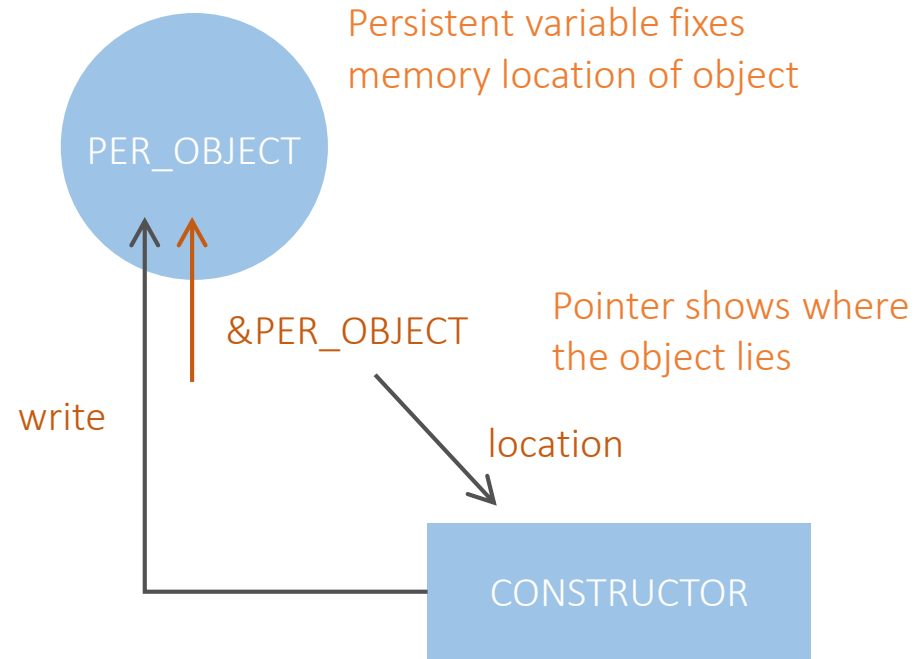
# 1 TESTING C++ CLASSES - CORE ELEMENTS



Unit testing of C++ code with testIDEA is essentially the same as C code testing with the exception that C++ objects need to be initialized prior to using their methods. These objects have to be allocated to a persistent variable in order to ensure that the constructor is called.



*Persistent variables have a **lifetime** that spans many test cases. They must be deleted at the end when no longer needed.*



*The core difference between C and C++ testing are associated with the initialization of the objects prior to testing the class's methods.*

# 1 TESTING C++ CLASSES - CODE

This unit uses for its examples a small demonstration application which includes the C++ Class *Temperature*.

Its purpose is to maintain a temperature value in three different units, Celsius, Fahrenheit and Kelvin (stored as 3 private values).

The class also has various public methods which allow us to operate and work on this data, including three different constructors.

```
class Temperature {
    private:
        float _tCelcius;
        float _tFahrenheit;
        float _tKelvin;
        float convCtoF(float temperature);
        float convCtoK(float temperature);
        float convFtoC(float temperature);
        float convFtoK(float temperature);
        float convKtoC(float temperature);
        float convKtoF(float temperature);

    public:
        float getTemperature();
        float getTemperature(tTypes type);
        void setTemperature(float temperature);
        void setTemperature(float temperature, tTypes
type);
        Temperature();
        Temperature(float temperature);
        Temperature(float temperature, tTypes type);
};
```

# 1 TESTING C++ CLASSES - CODE

The first constructor creates an object where Celsius is set to zero and Fahrenheit and Kelvin units are calculated appropriately.

The second constructor uses the value passed for *temperature* and assumes that value to be in Celsius. It then calculates Fahrenheit and Kelvin respectively.

The third constructor allows us to pass the temperature in a chosen unit that is defined by the second parameter.

```
Temperature::Temperature(void) {
    _tCelcius = 0.0;
    _tFahrenheit = convCtoF(_tCelcius);
    _tKelvin = convCtoK(_tCelcius);
}

Temperature::Temperature(float temperature) {
    _tCelcius = temperature;
    _tFahrenheit = convCtoF(temperature);
    _tKelvin = convCtoK(temperature);
}

Temperature::Temperature(float temperature, tTypes
type) {
    if (type == tTypes::tC) {
        Temperature(_tCelcius);
    } else if (type == tTypes::tF) {
        _tFahrenheit = temperature;
        _tCelcius = convFtoC(_tFahrenheit);
        _tKelvin = convFtoK(_tFahrenheit);
    } else {
        /* Assume Kelvin value */
        _tKelvin = temperature;
        _tCelcius = convKtoC(_tKelvin);
        _tFahrenheit = convKtoF(_tKelvin);
    }
}
```

# 1 TESTING C++ CONSTRUCTORS - CREATE BASE TEST



## Creating a base test for the constructor *Temperature(float)*

Having chosen the constructor we wish to test from the *Function* drop-down list, we see it actually has two parameters; the *temperature* we wish to pass into the constructor, and another parameter named *this*.

*this* is simply a pointer to the object in memory where all the data associated with the object (e.g. the private variables) are to be stored.

As we have not yet created an object for the constructor or other methods to use, we do not add any further information here in the wizard, instead finishing by clicking *Finish*.

**New test case wizard**

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:

Parameters:

**Expected result**

☐ Default expression for function return value test

☒ Custom expression and function return value name  
Expression:   
Ret. val. name:

< Back Next > Finish Cancel

# 1 TESTING C++ CONSTRUCTORS - CREATE BASE TEST



As we are creating a base test we will clear the *Execute* box in meta data form as we previously did for C unit tests.

The screenshot shows the testIDEA meta-data form for creating a base test. The form includes the following fields and controls:

- Execute:** A checkbox that is currently unchecked. A tooltip points to it with the text: "Check this box to enable the test. If unchecked, test will not be executed regardless of filters. It should be unchecked for test specifications, which are used as base for derived exceptions only, and are not intended for execution."
- Scope:** A radio button control, currently set to "Execute".
- ID:** A text field containing the value "vyjuzeratjk".
- Description:** A large text area with a "View / Edit" button to its right.
- Result comment:** A text area with a "View" button to its right. A tooltip points to it with the text: "This text refers to specific test run. It is stored to results and report only, and will be lost on next run!"
- Tags:** A text field.
- Log before:** A text field.
- Log after:** A text field.

On the right side of the form, there are several "Inherit" checkboxes and a "View" button with a pencil icon.

# 1 TESTING C++ CONSTRUCTORS - PERSISTENT VARIABLE



We add a persistent variable to the base test. We declare the variable within the base test and delete it immediately afterwards.

This will provide us with a *per\_Object* in each derived test which will be deleted upon test completion.

The screenshot shows the testIDEA interface. On the left, a sidebar contains a tree view with the following items: Meta, Function, Persistent variables (selected), Variables, Pre-conditions, Expected, Stubs, User Stubs, Test Points, Analyzer, Coverage, Profiler, Code areas, Data areas, Trace, HIL, Scripts, Options, Dry run, and Diagrams. The main window displays two tables: 'Declarations of persistent variables' and 'Deleted persistent variables'. Both tables have columns for 'Variable name' and 'Variable type'. The 'Declarations of persistent variables' table has one row with 'per\_Object' and 'Tem'. The 'Deleted persistent variables' table has one row with 'per\_Object'. A dropdown menu is open for the 'per\_Object' variable in the 'Declarations of persistent variables' table, showing a list of 'Temperature' related types and methods: Temperature, Temperature::Temperature(), Temperature::Temperature(float), Temperature::Temperature(float,tTypes), Temperature::convCtoF, Temperature::convCtoK, Temperature::convFtoC, Temperature::convFtoK, Temperature::convKtoC, and Temperature::convKtoF.

| Variable name | Variable type |
|---------------|---------------|
| per_Object    | Tem           |

| Variable name | Variable type |
|---------------|---------------|
| per_Object    |               |



# 1 TESTING C++ CONSTRUCTORS - CREATE DERIVED TEST



## Create a derived test

We have to use the *Test* option of the main menu bar once again and select *New derived test*.

The function name is inherited, so we don't have to fill this field.

Now we have to add the parameters: Obviously, we need a parameter for the temperature we want to pass (e.g. 0.0°C), but we also have to have a pointer to the temperature object to store this information.

Here we insert the pointer to the object, so that the parameter list is: *&per\_Object, 0.0*

New derived test case wizard

**New test case wizard**  
Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:

Temperature \* (Temperature \* this, float temperature)

Parameters:  Function parameters, for example: 10, 30, 'c'

**Expected result**

☐ Default expression for function return value test  
\_isys\_rv ==

☒ Custom expression and function return value name  
Expression:   
Ret. val. name:

< Back Next > **Finish** Cancel

# 1 TESTING C++ CONSTRUCTORS - CREATE DERIVED TEST



It is possible to displays the list and datatypes of the parameters once again in this dialogue by selecting and then immediately deleting the function to be testing from the field *Function*.

New derived test case wizard

**New test case wizard**

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:

Parameters:

Expected result

☐ Default expression for function return value test

☒ Custom expression and function return value name

Expression:

Ret. val. name:

< Back Next > Finish Cancel

# 1 TESTING C++ CONSTRUCTORS - CREATE DERIVED TEST



It is recommended to check the persistent variables once again after creating the derived test to ensure that the data has been inherited correctly.

The screenshot shows the testIDEA interface. On the left, a sidebar contains a tree view with the following items: Meta, Function, Persistent variables (selected), Variables, Pre-conditions, Expected, Stubs, User Stubs, Test Points, Analyzer (expanded), Coverage, Statistics, Profiler, Code areas, Data areas, Trace, HIL, Scripts, Options, Dry run, and Diagrams. The main window displays two tables:

**Declarations of persistent variables**

|   | Variable name | Variable type |
|---|---------------|---------------|
| 0 | per_Object    | Temperature   |

**Deleted persistent variables**

|   | Variable name |
|---|---------------|
| 0 | per_Object    |

# 1 TESTING C++ CONSTRUCTORS - CREATE DERIVED TEST

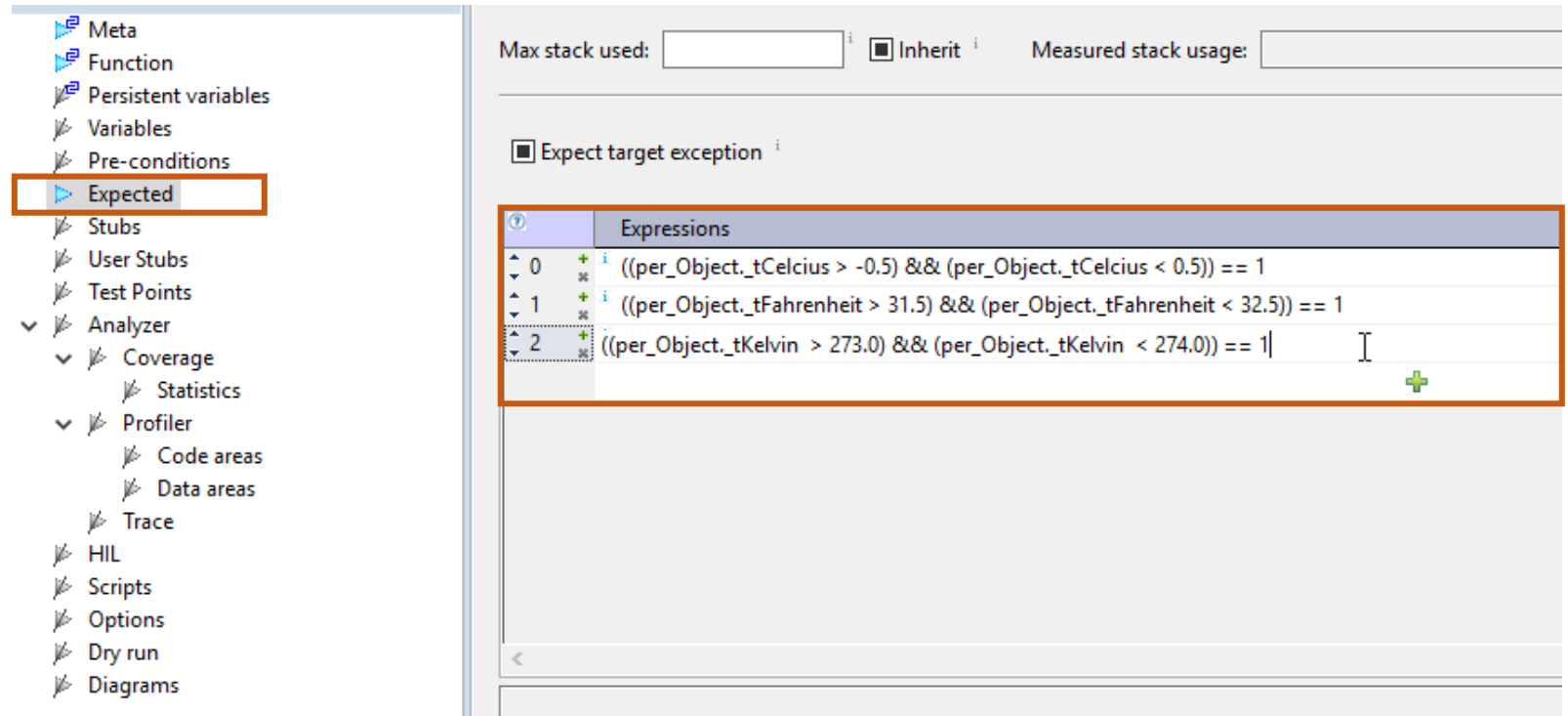


We now have a structure for our test cases but we don't have any expected values to define the pass/fail criteria.

As the constructor does not return a result we instead need to check that the object (specifically its private variables) were correctly initialized. In the *Expected* field (as shown here) we have defined ranges for the expected results that we consider acceptable for a pass result.

Expressions for the *Expected* form can take any valid C/C++ form.

The private variables of the class can be accessed using dot-notation, i.e. `per_Object._tCelcius`.



*Normally, private members of a class are not accessible. However, via testIDEA, we have full access to the inner workings of classes. This is what enables us to test such code.*

# 1 TESTING C++ CONSTRUCTORS - CREATE DERIVED TEST



## Working with float values

For the purposes of testing, float values shouldn't be compared for strict equality.

Here we have chosen to test for a  $\pm 0.5^\circ\text{C}$  range around the desired float value allows for small errors caused by the datatype's representation limitations to be ignored.

The screenshot shows the testIDEA interface. On the left, a sidebar contains a tree view with the following items: Meta, Function, Persistent variables, Variables, Pre-conditions, **Expected** (highlighted with an orange box), Stubs, User Stubs, Test Points, Analyzer, Coverage, Profiler, Trace, HIL, Scripts, Options, Dry run, and Diagrams. The main window displays the 'Expected' tab, which includes a table of expressions. The table has a header 'Expressions' and three rows of test expressions, each with a green plus icon in the first column. The expressions are: 0: `((per_Object._tCelcius > -0.5) && (per_Object._tCelcius < 0.5)) == 1`, 1: `((per_Object._tFahrenheit > 31.5) && (per_Object._tFahrenheit < 32.5)) == 1`, and 2: `((per_Object._tKelvin > 273.0) && (per_Object._tKelvin < 274.0)) == 1`. The third row is highlighted with an orange box. Above the table, there are input fields for 'Max stack used:' and 'Measured stack usage:', and a checkbox labeled 'Expect target exception' which is checked.

# 1 TESTING C++ CONSTRUCTORS - CREATE A FEW MORE TESTS



Further tests are created with ease by using the *Table* view mode.

The easiest way to add further test vectors is to copy the parameters and the expected values of existing tests, changing the input parameters and expected results manually.

There is now a suite of tests that can be used to check the functionality of one of the class's constructors.

Tests for the remaining constructors can be created in a similar manner.

| ① | func                            |             |        |        |  |  |  |  |
|---|---------------------------------|-------------|--------|--------|--|--|--|--|
|   | func                            | params      |        | retVal |  |  |  |  |
|   |                                 | 0           | 1      |        |  |  |  | 0  |
| 0 | Temperature::Temperature(float) |             |        |        |  |  |  |  |
| 1 | Temperature::Temperature(float) | &per_Object | 0.0    |        |  |  |  | ((per_Object_tCelcius > -0.5) && (per_Object_tCelcius < 0.5)) == 1     |
| 2 | Temperature::Temperature(float) | &per_Object | 10.8   |        |  |  |  | ((per_Object_tCelcius > 10.0) && (per_Object_tCelcius < 11.0)) == 1    |
| 3 | Temperature::Temperature(float) | &per_Object | 95.0   |        |  |  |  | ((per_Object_tCelcius > 94.5) && (per_Object_tCelcius < 95.5)) == 1    |
| 4 | Temperature::Temperature(float) | &per_Object | -1.0   |        |  |  |  | ((per_Object_tCelcius > -1.5) && (per_Object_tCelcius < -0.5)) == 1    |
| 5 | Temperature::Temperature(float) | &per_Object | -50.3  |        |  |  |  | ((per_Object_tCelcius > -51.0) && (per_Object_tCelcius < -50.0)) == 1  |
| 6 | Temperature::Temperature(float) | &per_Object | -270.0 |        |  |  |  | ((per_Object_tCelcius > -270.5) && (per_Object_tCelcius < 269.5)) == 1 |

## 2 TESTING A C++ METHOD



In the previous test we only tested whether the constructor is constructing the object properly and if the private variables have been initialized correctly.

Now we want to test the class's *getTemperature()* method.

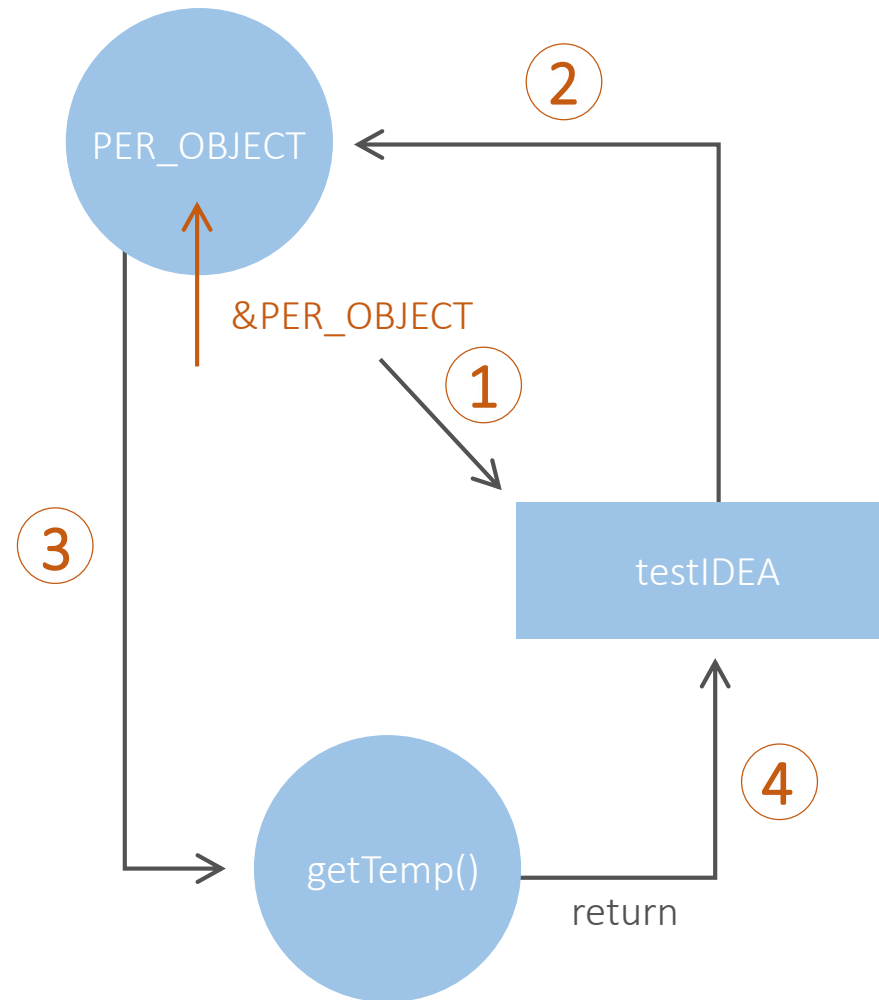
```
float Temperature::getTemperature() {  
    return _tCelcius;  
}  
  
float Temperature::getTemperature(tTypes type) {  
    float returnValue = 0.0;  
  
    if (type == tTypes::tC) {  
        returnValue = _tCelcius;  
    } else if (type == tTypes::tF) {  
        returnValue = _tFahrenheit;  
    } else {  
        /* Assume Kelvin value */  
        returnValue = _tKelvin;  
    }  
    return returnValue;  
}
```

## 2 TESTING A C++ METHOD - PROCESS



The process is very similar to that used to develop tests for C code but with the additional of a single extra step – the constructor needs to have been called before we can test any class methods.

The function *getTemperature()* simply returns the current temperature. Thus the test case can simply compare the returned value from the method with the value expected. The value expected, however, will depend on the current state of the internal private variables of the class – this will likely depend on the state they were left in after the previous test.

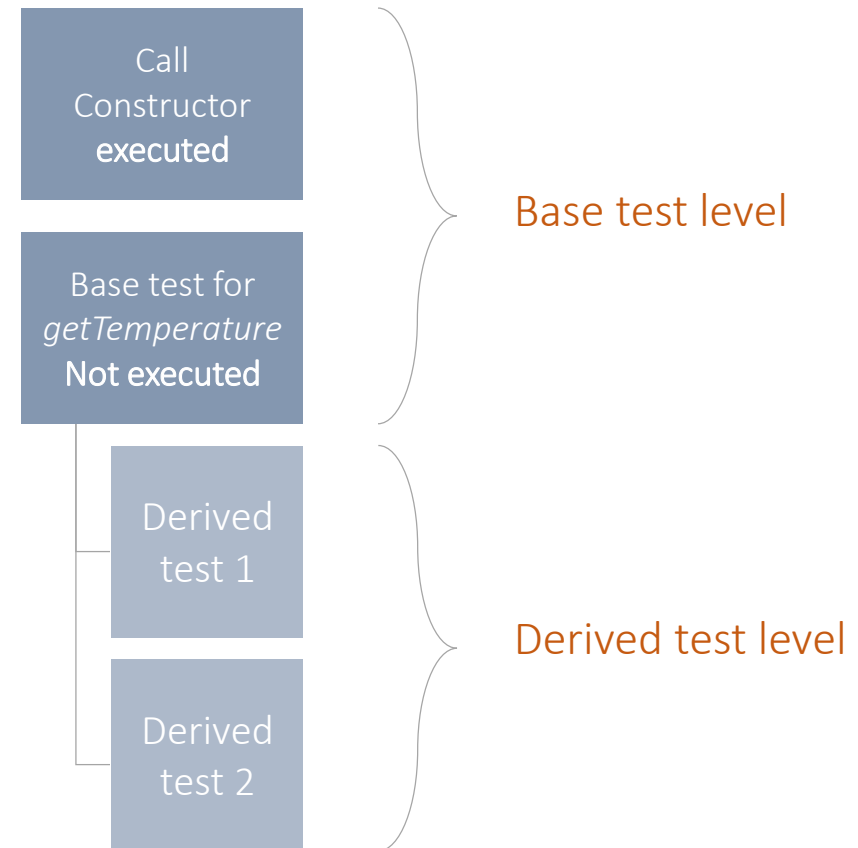




## 2 TESTING A C++ METHOD - TESTING CONCEPT



To test a C++ method it is recommended to work with the displayed structure of base and derived tests.



## 2 TESTING A C++ METHOD - CALL CONSTRUCTOR



For the **first test** we have to **call the C++ constructor** for the class. This is performed at base test level.

In this case we will use the constructor  
`Temperature::Temperature(float).`



*This test case is not a base test but a regular test vector at base test level. So the **Execute** box in the meta data has to be left **Enabled**.*

New test case wizard

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:

Parameters:

Expected result

☐ Default expression for function return value test  
\_sys\_rv ==

☒ Custom expression and function return value name  
Expression:   
Ret. val. name:

< Back Next > Finish Cancel

# 2 TESTING A C++ METHOD - CREATE & DELETE PERSISTENT VARIABLE

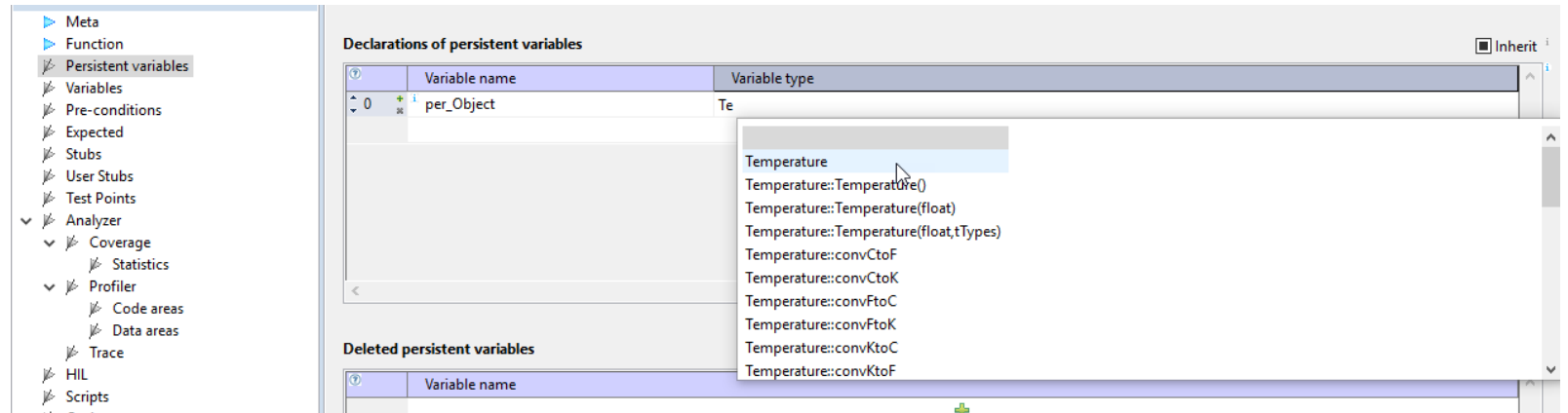


When testing a C++ method we again need to ensure there is a persistent variable *per\_Object* created for use over the duration of testing. In the **first test that is to be executed** (in this case, the class constructor test) we define a variable of type *Temperature*.

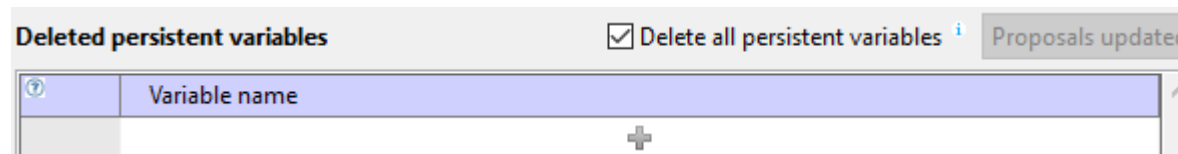
In the **very last derived test** we must **delete** the persistent variable again.

In this way we create a class object which will exist for the entire time of testing. The object's state is retained between tests and modified corresponding to the tests executed.

First executed test vector (call of class constructor):



Very last test vector (last derived test for *getTemperature()*):

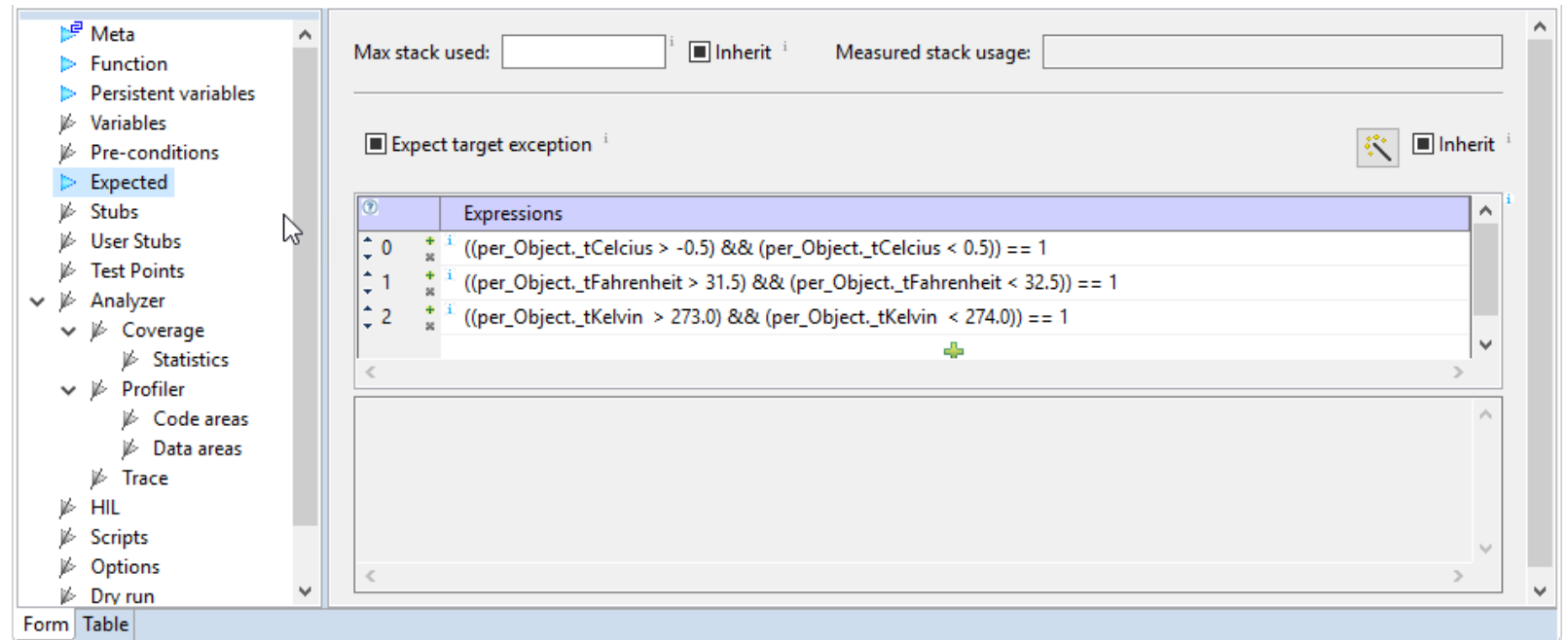


## 2 TESTING A C++ METHOD - TEST CONSTRUCTOR (OPTIONAL)



**Optional:** You may wish to test that the constructor initialized the object correctly by making sure that the internal values of the object correspond to 0.0°C.

If you have already tested the constructor elsewhere, this is not absolutely necessary. However, if you wish to include this pass/fail criteria again, it can be added as shown here.



## 2 TESTING A C++ METHOD - CREATE A BASE TEST FOR THE METHOD



Next we will configure the base test for the `getTemperature()` method using the *New test case wizard*.

In the base test the parameter `&per_Object` is entered.

This reference to the parameter `&per_Object` is entered in the base test to enable inheritance of this parameter into all of the derived tests.

Once this base test is created (after clicking *Finish*), remember to clear the check box for the *Execute* option.

**New test case wizard**

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:

Parameters:  Function parameters, for example: 10, 30, 'c'

**Expected result**

☐ Default expression for function return value test  
\_sys\_rv ==

☒ Custom expression and function return value name  
Expression:   
Ret. val. name:

## 2 TESTING A C++ METHOD - CREATE A DERIVED TEST



The next step requires the creation of an executable derived test from the base test. Rather than enter information into the wizard, we have chosen here to enter the pass/fail criteria directly into the form fields.

After the initialization of the object with the default constructor, we expect 0.0°C as our return value for this first test.

In the *Expected* form we will again define a small acceptable range as our pass/fail criteria.

Max stack used:  ☒ Inherit Measured stack us

☒ Expect target exception

|   | Expressions   |
|---|---|
| 0 | <code>((_isys_rv &gt; -0.5) &amp;&amp; (_isys_rv &lt; 0.5)) == 1</code> |

# 2 TESTING A C++ METHOD - DELETE PERSISTENT VARIABLE

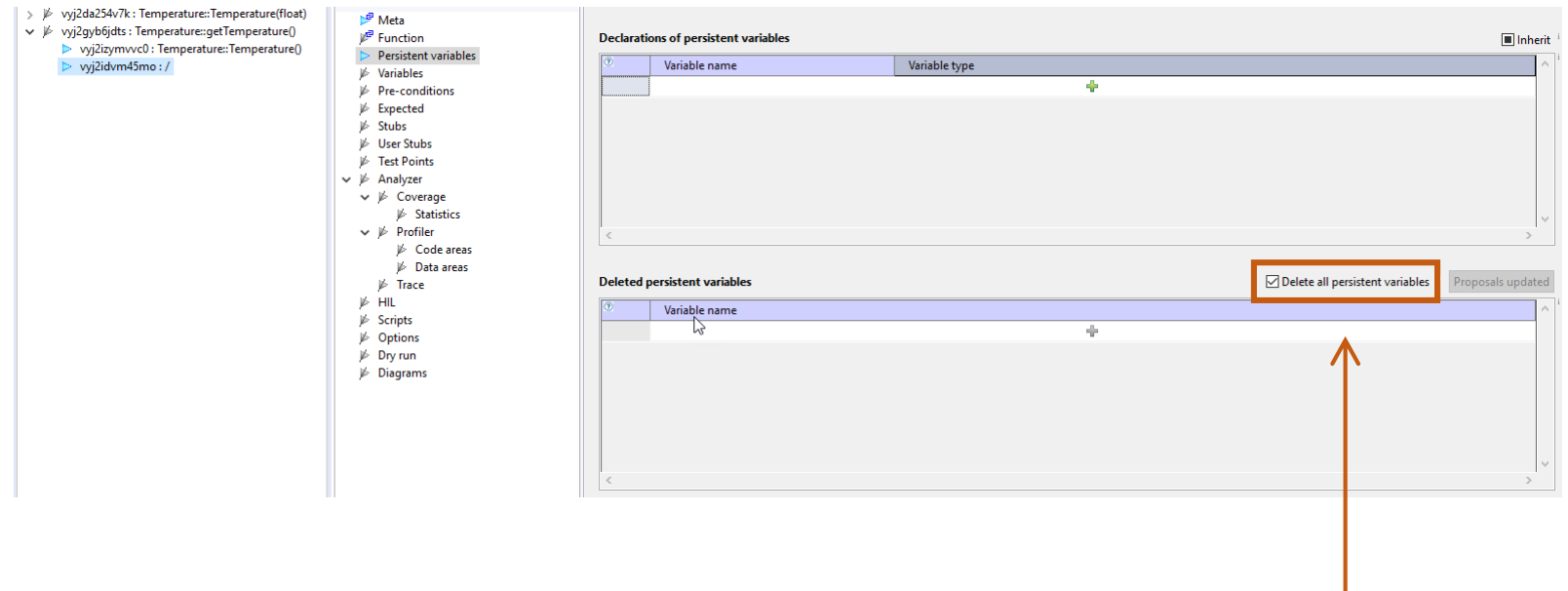


Remember:

Ensure that the persistent variable is **deleted** at the end of the final derived test. To do this, select the last test in the group and configure it to delete all persistent variables at the end of testing.

This is really the key difference between testing C++ and C:

A class object must be created as a persistent variable that exists across several different tests, and variable needs to be deleted again upon completion of testing.



Now we are ready to execute our tests!

## SUMMARY

---

# testIDEA



## C++ constructors and methods - differences in testing

| C++ CONSTRUCTOR                                    | C++ METHOD   |
|--|--|
| Define and delete persistent variable in each test | Define persistent variable in first executed test (constructor call) |
|  | Delete persistent variable in final executed derived test            |

