

**Module:** Fundamentals of Test Automation

**Student:** Leonardo de Melo Abreu

**Professor:** Breno Miranda

Link for the video: [https://drive.google.com/file/d/1\\_sRAHy4WWRPJnl\\_ZNNdhKaNn-raE9j-T/view?usp=sharing](https://drive.google.com/file/d/1_sRAHy4WWRPJnl_ZNNdhKaNn-raE9j-T/view?usp=sharing)

## Tutorial

### Overview

In this tutorial, we will learn about unit testing, its importance, and how to create tests in Visual Studio 2022. We will also explore practical examples of creating tests to ensure the correct functionality of the code for a calculator.

### What are Unit Tests?

At its core, unit tests are a powerful tool that allows us to thoroughly examine specific parts of our code. Think of them as magnifying lenses that focus on libraries, classes, functions, and various elements of the program. The underlying goal is to determine if the code is not only operational but also operating accurately and as expected for a well-defined set of inputs.

Unit tests go beyond merely checking if the software works. They are designed to explore the nooks and crannies of functionalities, exposing any discrepancies between actual behavior and the desired one. By focusing on distinct code fragments, we ensure that each part contributes to the overall functionality, preventing slip-ups that could go unnoticed in broader tests.

### CppUnitTest

CppUnitTest is a unit testing tool for C++ that is part of Visual Studio. It provides a framework for writing and executing unit tests, as well as visualizing test results.

### Microsoft Visual Studio 2022

Considering that CppUnitTest is present in the Microsoft Visual Studio 2022 environment, we will walk through the step-by-step process of creating unit tests using this tool. The intention is for this tutorial to illustrate concepts and demonstrate the necessary configurations for developing unit tests through Microsoft Visual Studio 2022.

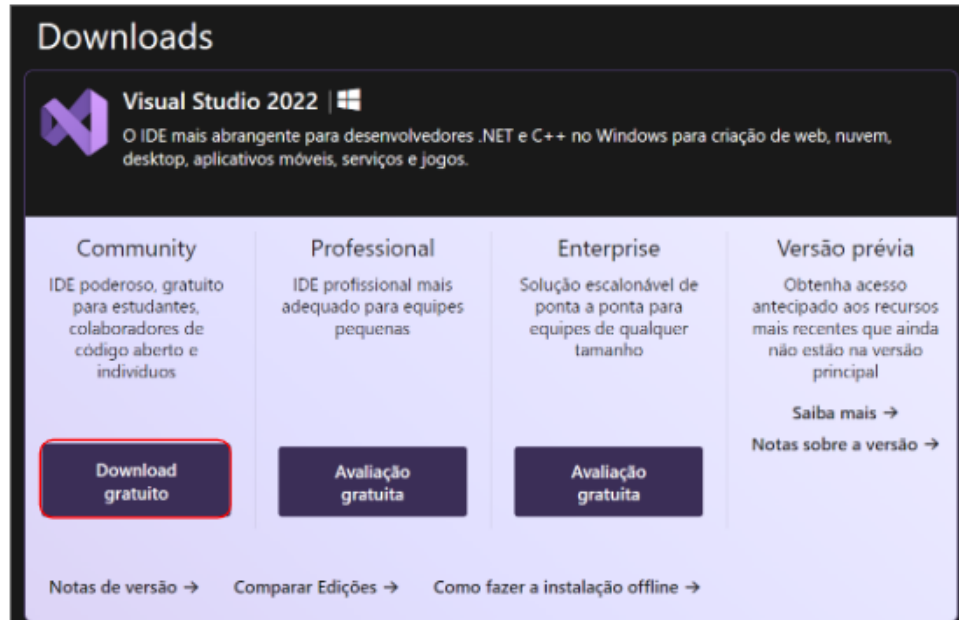
### Getting Started

**Step 1** -Install Microsoft Visual Studio 2022.

Disponível em: <https://visualstudio.microsoft.com/pt-br/downloads/>

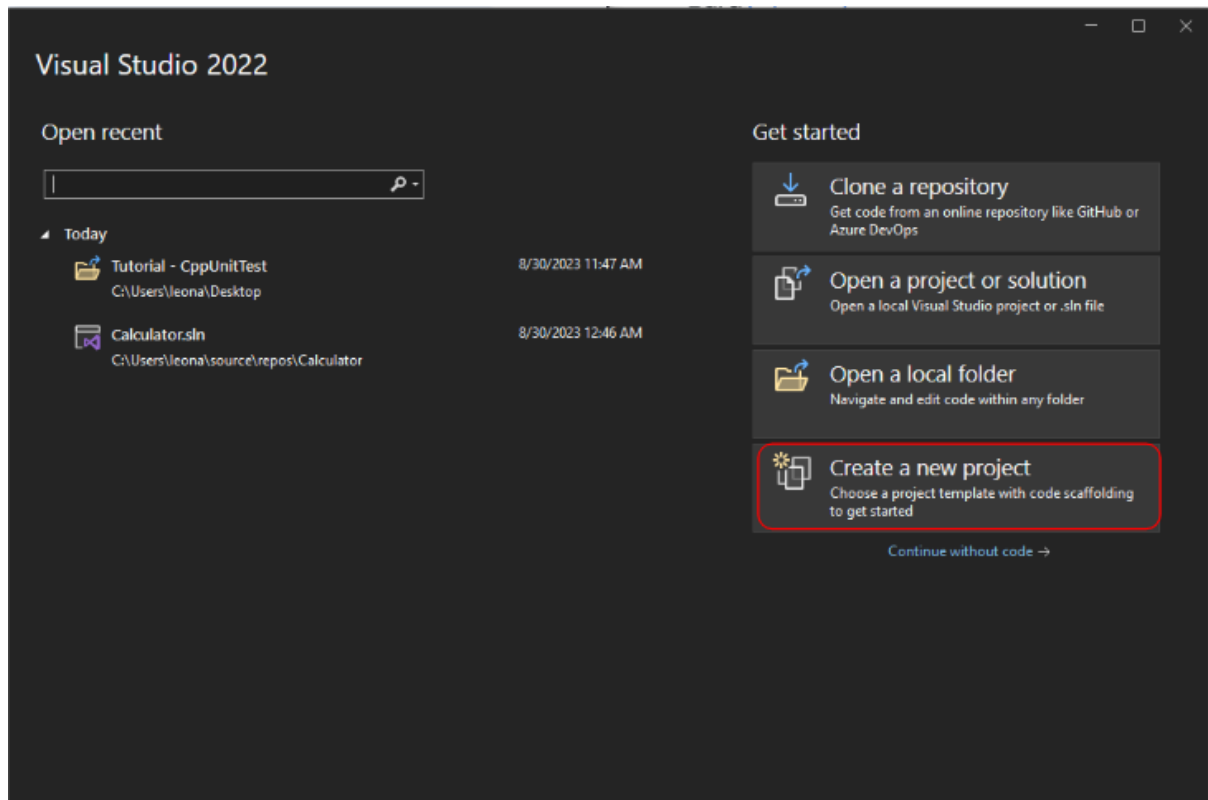
Available at: <https://visualstudio.microsoft.com/en/downloads/>

The software can be downloaded for free by clicking the "Free Download" button.

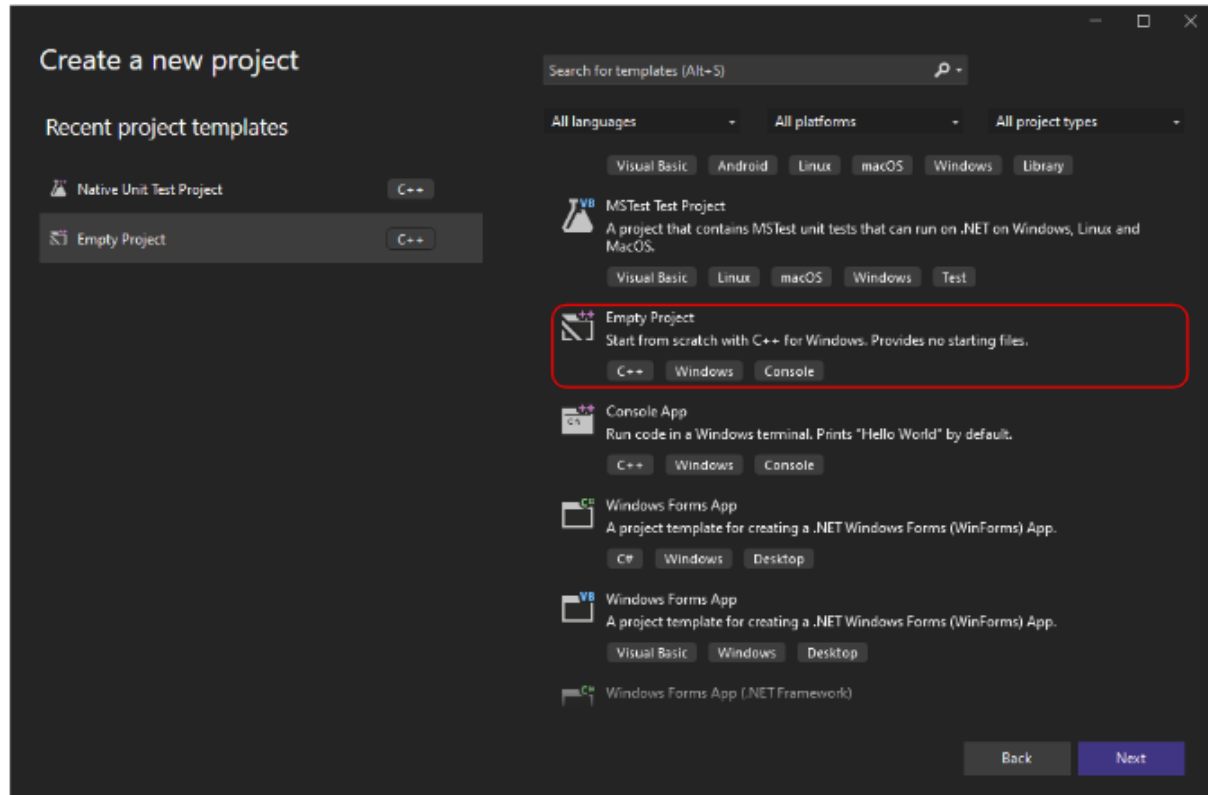


**Note:** The installation process is quite straightforward; you simply need to run the downloaded executable file and follow the recommended installation instructions.

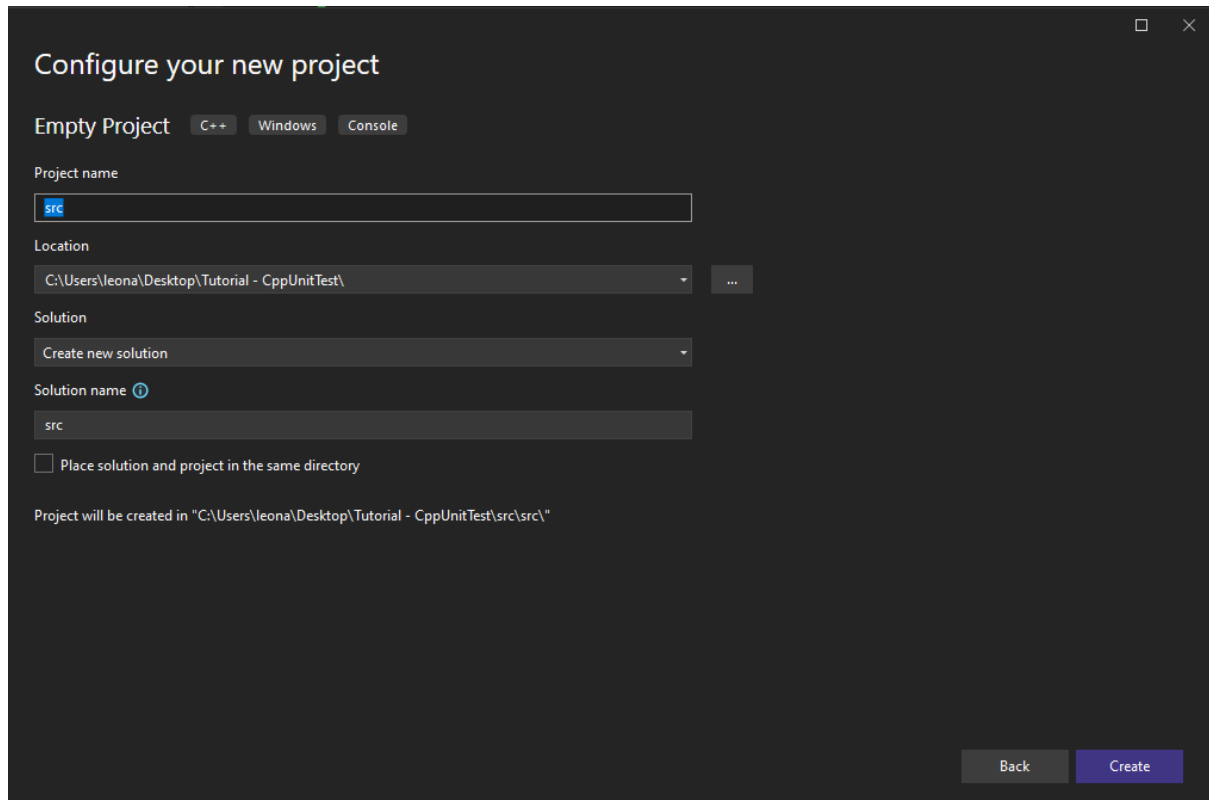
**Step 2** - With Microsoft Visual Studio 2022 already installed, you will need to create a new project. To do this, open Microsoft Visual Studio 2022. On the start page, click on "Create a new project."



Then, select the "Empty Project" option in the tab that opens, and click the "Next" button.

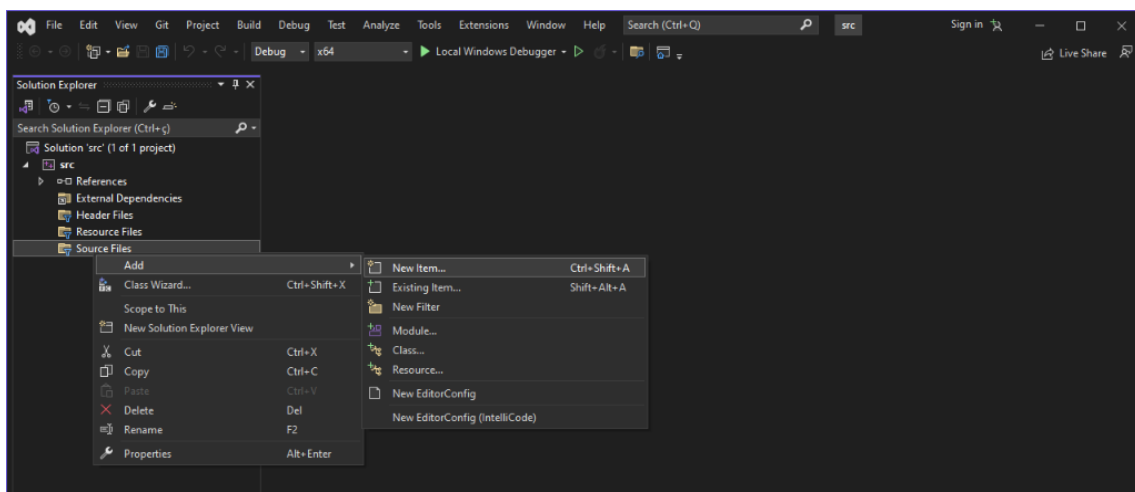


Now, simply enter the project name in the "Project name" field and click the "Create" button.

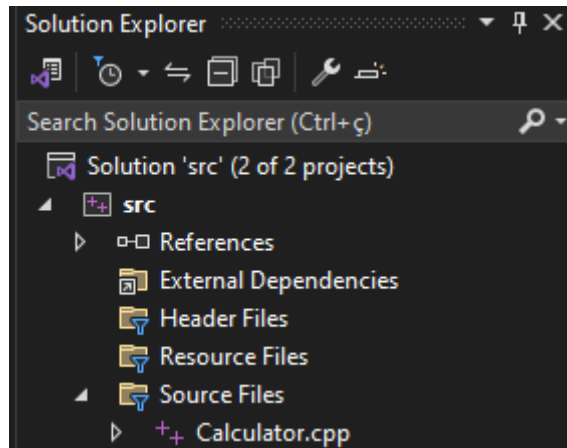


**Step 3** - Now that we have created the project, we need to create the file that will contain the code we will use for our tests.

To do this, right-click on the "Source Files" folder. When you do this, a new window will open with several options. Select the "Add" option, and then click on "New Item."



Now you can see the file with the chosen name in the "Solution Explorer" menu, along with the project folders. For this example, a file named "Calculator.cpp" has been created.



Clicking on the "Calculator.cpp" file will open the text editor for typing in the code. In the following section, you'll find the example code that was used as the basis for this tutorial:

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

int soma(int a, int b) {
    return a + b;
}

int sub(int a, int b) {
    return a - b;
}

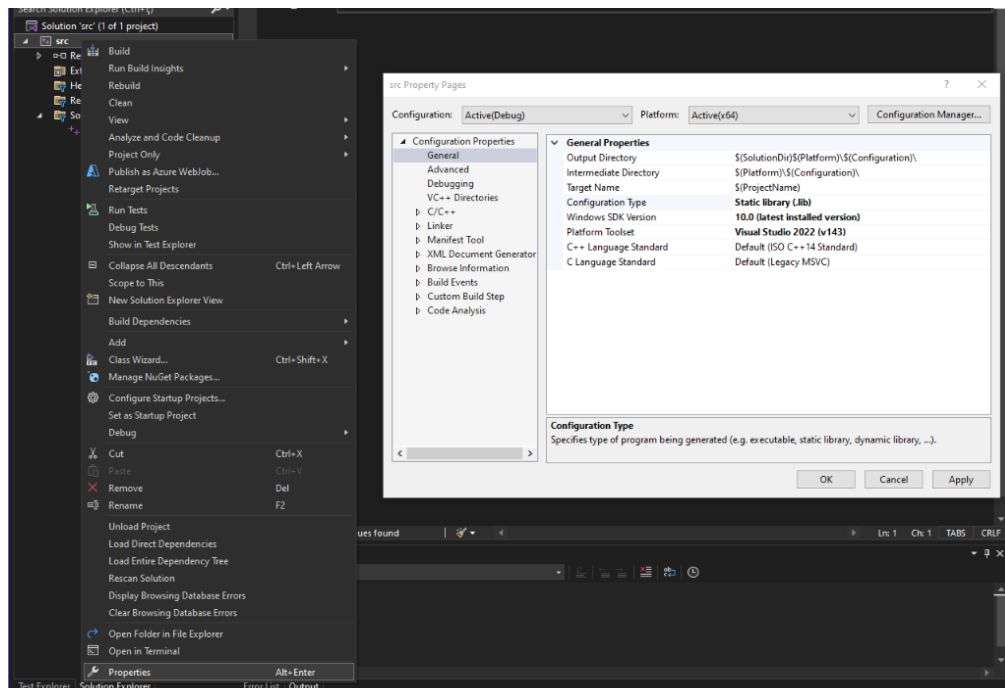
int mult(int a, int b) {
    return a * b;
}

int divide(int a, int b)
{
    if (b == 0)
    {
        throw std::invalid_argument("Divisão por zero."); // Lança uma exceção
    }

    return a / b;
}
```

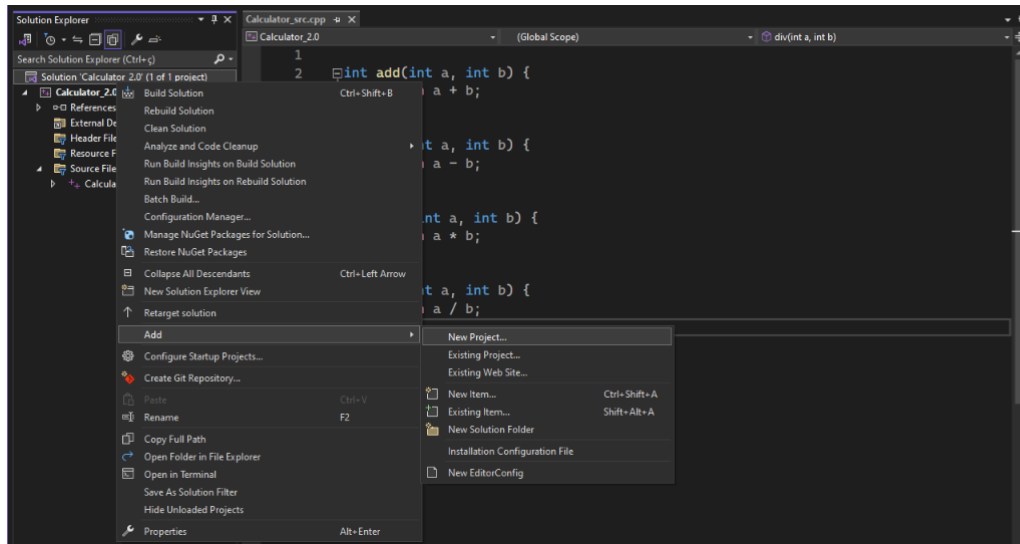
The next step to take is to configure the "Calculator.cpp" file as a library so that we can easily use it to test the functions implemented in the project that will be created later with the test cases.

You can do this by right-clicking on the project name, in this case, "src". A menu with various options will appear; click on "Properties." A window will open. Click on "Configuration Properties" -> "General" and change the "Configuration Type" to "Static library (.lib)" and click "Apply".

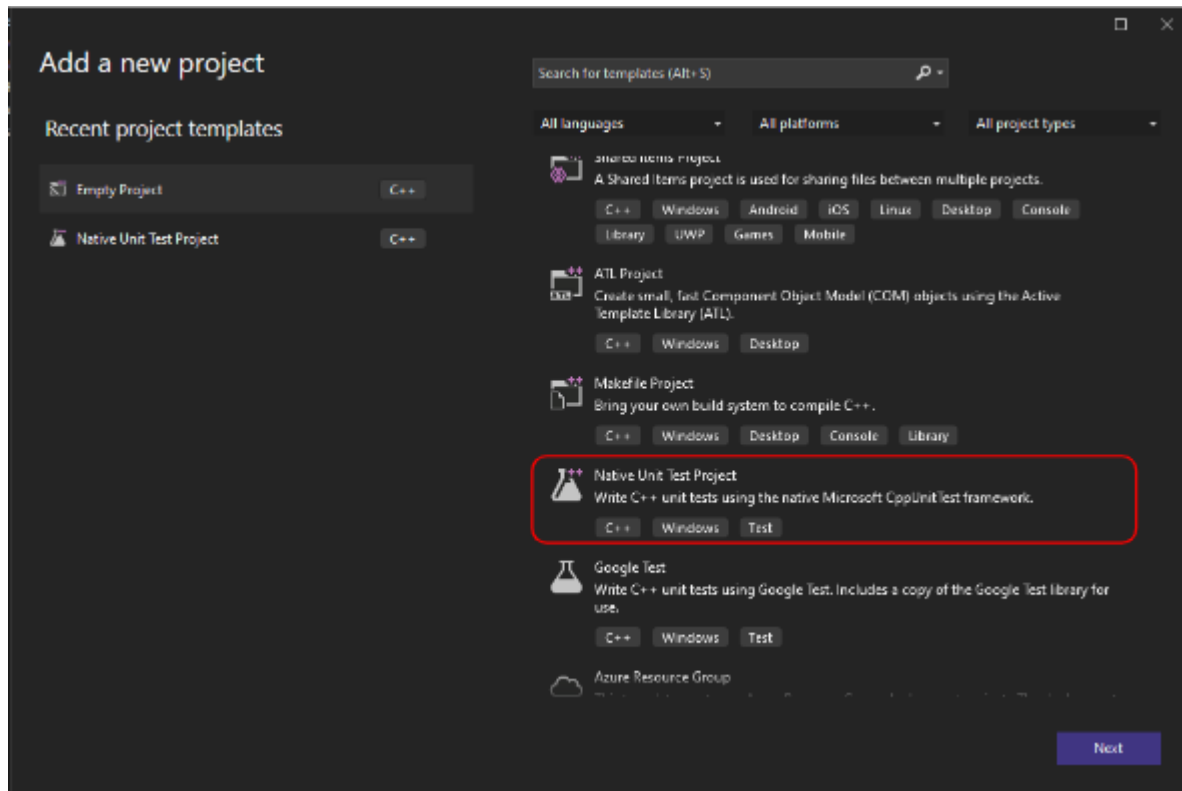


**Step 4 - Great! Now that we have the first project set up, let's create the project that will be responsible for containing the test cases.**

To do that, click on "Solution <project name>." An auxiliary tab will open. Click on the "Add" option -> "New Project".

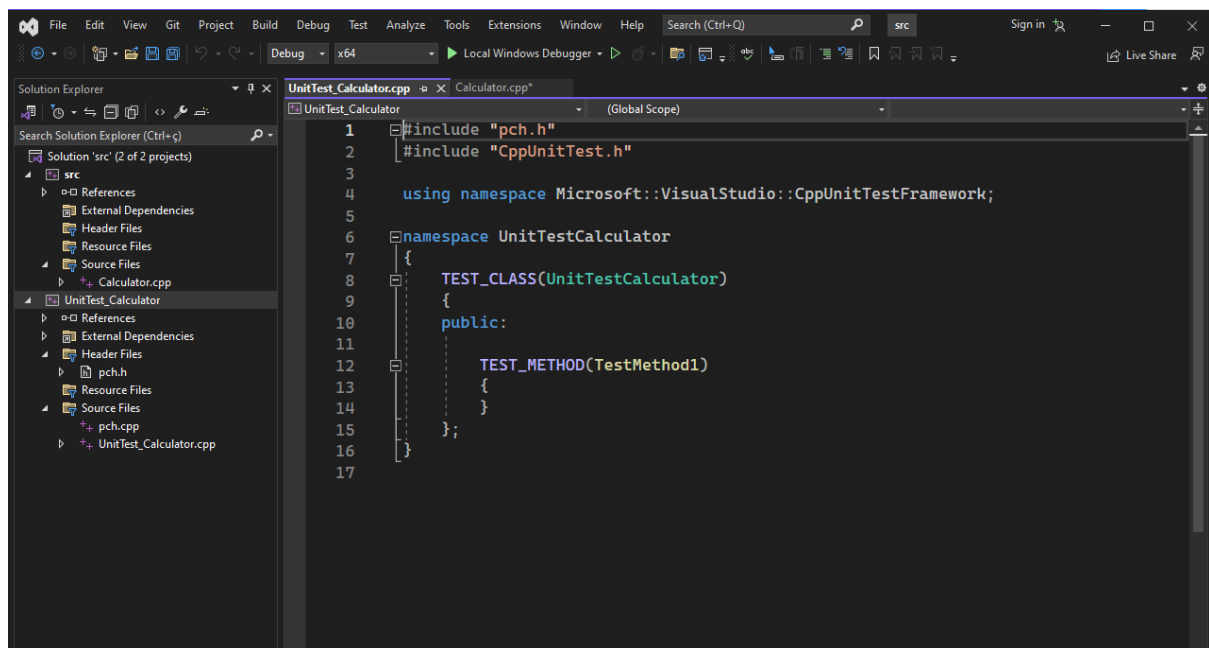


When you click on "New Project," the following window will open:



In this window, we will create a project for unit testing using Microsoft's CppUnitTest framework.

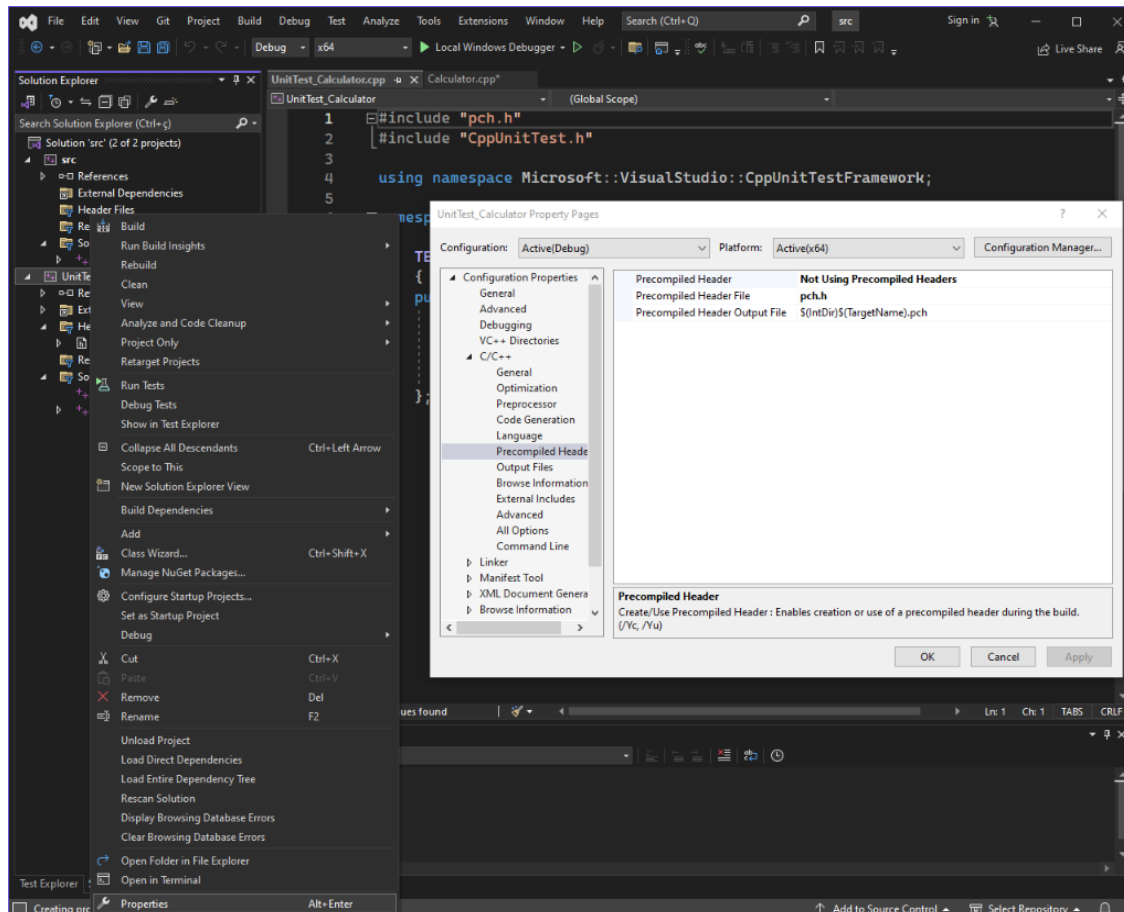
In the following figure, you can see that the project has been created successfully. When the project is created, the tool itself generates a structure that serves as a foundation for test development.



Upon creating the project, you'll notice that "pch.cpp" and "pch.h" files are generated. These are default files that provide a basic structure to start writing code. However, we won't be

using them. Hence, you can delete these files and request the system not to use these precompiled files.

To do this, right-click on the name of the test project. A tab will open; click on "Properties" -> "Configuration Properties" -> "Precompiled Header," and then select "<Not Using Precompiled Headers>" in the "Precompiled Header" field.



**Step 5** - With the file configured, you can now start writing the test cases. For this tutorial, the following test cases were written:

```
#include "Calculator.cpp" // Inclui o arquivo de implementação da calculadora
#include "CppUnitTest.h" // Inclui a biblioteca do CppUnitTestFramework

using namespace Microsoft::VisualStudio::CppUnitTestFramework; // Espaço de nomes para asserções do
//Visual Studio

namespace UnitTestCalculator // Namespace para os testes da calculadora
{
    TEST_CLASS(Soma) // Classe de testes para a função de soma
    {
    public:

        TEST_METHOD(Zero) // Método de teste para a soma com valores zero
        {
            Assert::IsTrue(soma(0, 0) == 0); // Verifica se a soma de 0 + 0 é igual a 0
        }
    }
}
```



```
        Assert::IsTrue(soma(0, 1) == 1); // Verifica se a soma de 0 + 1 é igual a 1
    }
    TEST_METHOD(Básico) // Método de teste para somas básicas
    {
        Assert::IsTrue(soma(2, 5) == 7); // Verifica se a soma de 2 + 5 é igual a 7
        Assert::IsTrue(soma(-2, 5) == 3); // Verifica se a soma de -2 + 5 é igual a 3

        Assert::AreEqual(soma(2, 5), 7); // Verifica se a soma de 2 + 5 é igual a 7
        Assert::AreEqual(soma(-2, 5), 3); // Verifica se a soma de -2 + 5 é igual a 3
    }
    TEST_METHOD(Comutativa) // Método de teste para a propriedade comutativa da soma
    {
        Assert::AreEqual(soma(2, 5), soma(5, 2)); // Verifica se a soma de 2 + 5 é igual à
//soma de 5 + 2
    }
    TEST_METHOD(Fal) // Método de teste para verificar falso
    {
        Assert::IsFalse(soma(0, 0) == 1); // Verifica se a soma de 0 + 0 não é igual a 1
    }
};

TEST_CLASS(Subtração) // Classe de testes para a função de subtração
{
public:
    TEST_METHOD(Zero) // Método de teste para subtrações com valor zero
    {
        Assert::IsTrue(sub(0, 0) == 0); // Verifica se a subtração de 0 - 0 é igual a 0
        Assert::AreEqual(sub(0, 5), -5); // Verifica se a subtração de 0 - 5 é igual a -5
    }

    TEST_METHOD(Básico) // Método de teste para subtrações básicas
    {
        Assert::IsTrue(sub(3, 4) == -1); // Verifica se a subtração de 3 - 4 é igual a -1
        Assert::IsTrue(sub(-2, 5) == -7); // Verifica se a subtração de -2 - 5 é igual a -7
    }

    TEST_METHOD(False) // Método de teste para verificar falso
    {
        Assert::IsFalse(sub(0, 0) == 1); // Verifica se a subtração de 0 - 0 não é igual a 1
    }

    TEST_METHOD(SelfSub) // Método de teste para subtração consigo mesmo
    {
        Assert::IsTrue(sub(2, 2) == 0); // Verifica se a subtração de 2 - 2 é igual a 0
    }
};

TEST_CLASS(Multiplicação) // Classe de testes para a função de multiplicação
{
public:
    TEST_METHOD(Zero) // Método de teste para multiplicações com valor zero
    {
        Assert::IsTrue(mult(0, 0) == 0); // Verifica se a multiplicação de 0 * 0 é igual a 0
        Assert::IsTrue(mult(0, 5) == 0); // Verifica se a multiplicação de 0 * 5 é igual a 0
    }
    TEST_METHOD(Básico) // Método de teste para multiplicações básicas
    {
        Assert::IsTrue(mult(3, 4) == 12); // Verifica se a multiplicação de 3 * 4 é igual a 12
        Assert::IsTrue(mult(-2, 5) == -10); // Verifica se a multiplicação de -2 * 5 é igual a
//-10
    }
    TEST_METHOD(Comutativa) // Método de teste para a propriedade comutativa da
//multiplicação
```

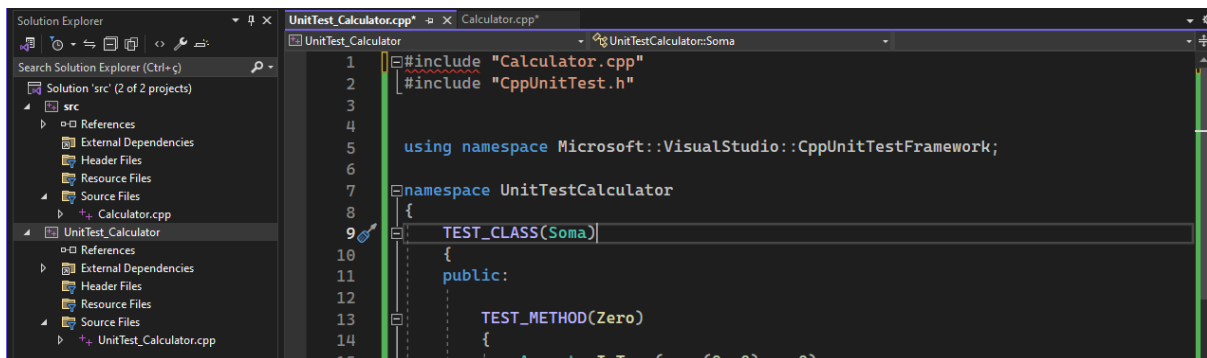
```

    {
        Assert::AreEqual(mult(2, 5), mult(5, 2)); // Verifica se a multiplicação de 2 * 5 é
//igual à multiplicação de 5 * 2
    }
    TEST_METHOD(False) // Método de teste para verificar falso
    {
        Assert::IsFalse(mult(0, 0) == 1); // Verifica se a multiplicação de 0 * 0 não é igual a
//1
    }
};
TEST_CLASS(Divisão) // Classe de testes para a função de divisão
{
public:
    TEST_METHOD(divi) // Método de teste da divisão básica
    {
        Assert::AreEqual(divide(10, 5), 2); // Compara o resultado da divisão com o valor
//esperado
    }
    TEST_METHOD(zero) // Método de teste para a divisão por zero
    {
        // Tenta chamar a função de divisão com um segundo argumento igual a zero
        try {

            divide(10, 0); // Chama a função de divisão com um segundo
//argumento igual a zero

            Assert::Fail(); // Se a função não lançar uma exceção, o teste falha
        }
        // Se a função lançar uma exceção, verifica se a mensagem da exceção é a
//esperada
        catch (std::invalid_argument const& e) {
            Assert::AreEqual(e.what(), "Divisão por zero.");
        }
    }
};
}

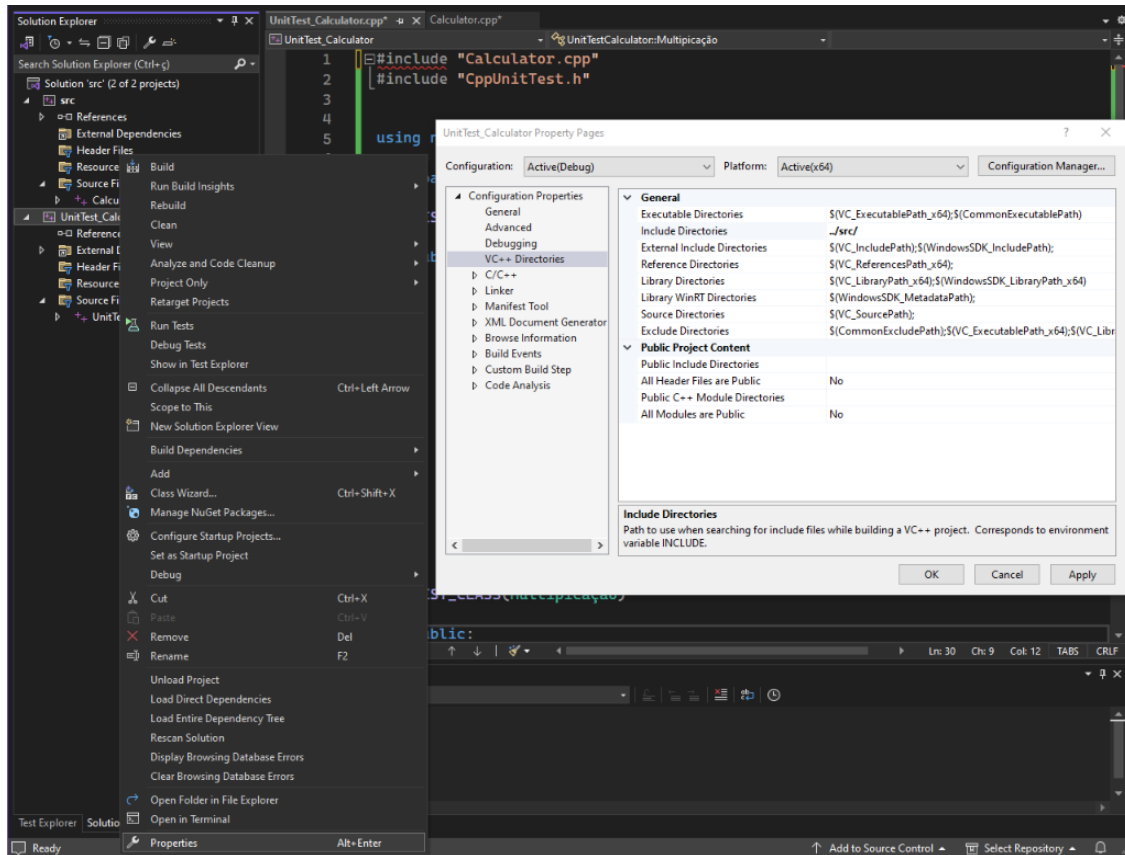
```



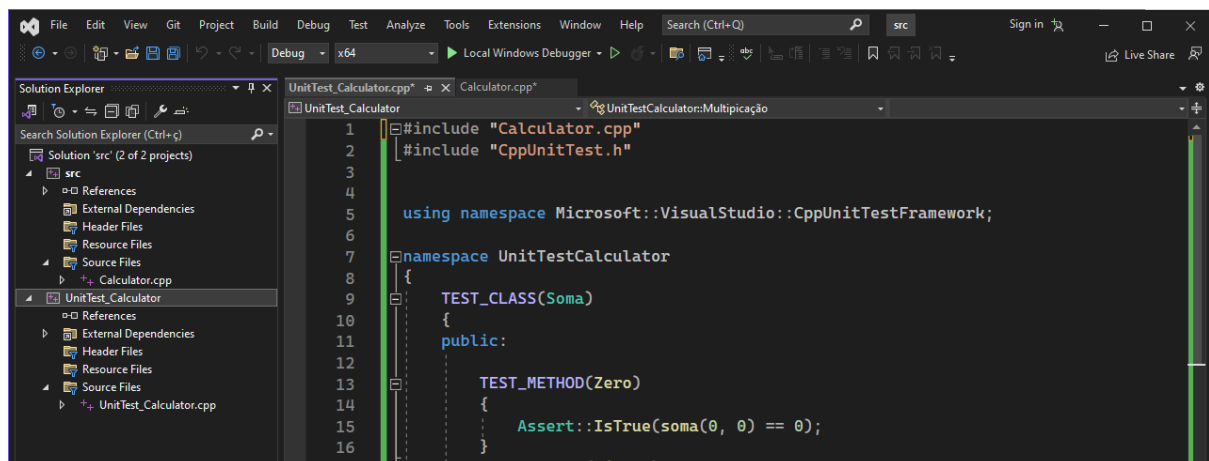
It's evident that when you insert the code, the program can't find the library where the required functions are located to execute the tests. This is because it's unable to locate the functions that are in another project, in another file. To make it recognize them, we need to provide the path so that the software can find the file with the functions.

This can be done by right-clicking on the test cases project folder, in this case, the "UnitTest\_Calculator" folder -> "Properties" -> "Configuration Properties" -> "VC++ Directories," and then add the "<Directory of the folder where the library you want to test is

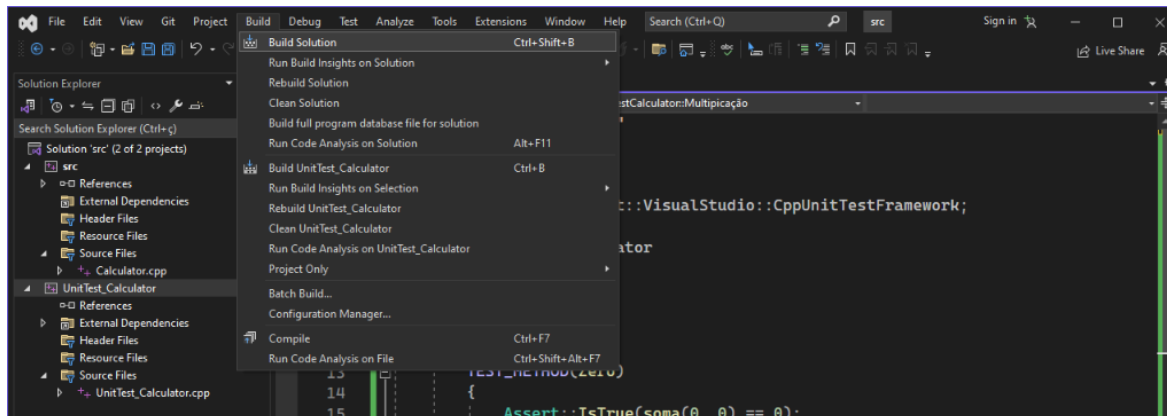
located>". If you're following this tutorial, simply navigate back one directory and enter the "src" directory.



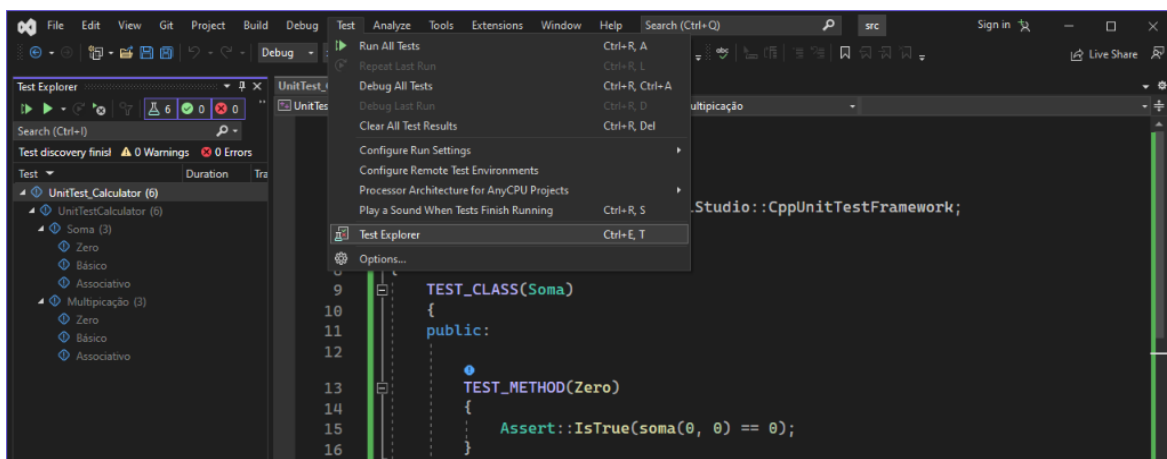
As you can see in the image below, after performing the procedure, the error mark disappears from the code, and now you can run the tests.



**Step 6** - To compile the files, you can click on the "Build" option and then select "Build Solution."

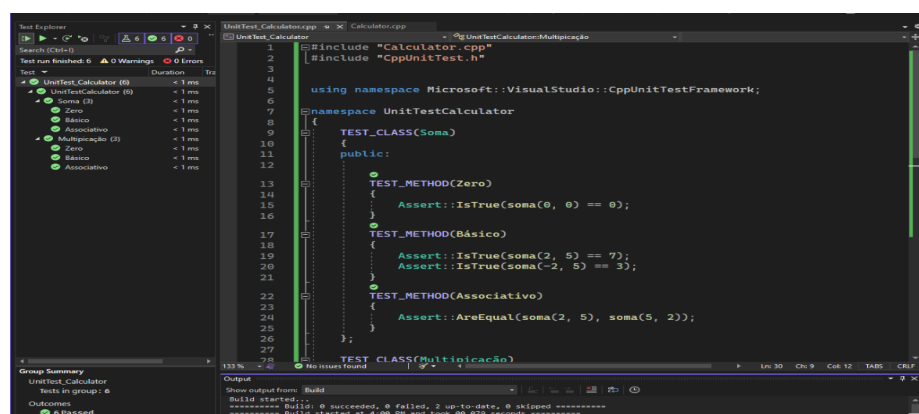


To analyze the tests, you can open the "Test Explorer," which can be found in the top menu by clicking on the "Build" tab -> "Test Explorer."



Now, with the "Test Explorer" open, when you click the "Run" button, the tests will be executed and displayed. When a test "passes," it turns green, and when a test "fails," it turns red.

In the image below, you can see the results for the tests that were mentioned earlier.



### Methods Used for this Tutorial:

Assert::IsTrue()	Checks if an expression is true.
Assert::IsFalse()	Checks if an expression is false.
Assert::AreEqual()	Checks if two values are equal.
Assert::Fail()	Fails the current test.