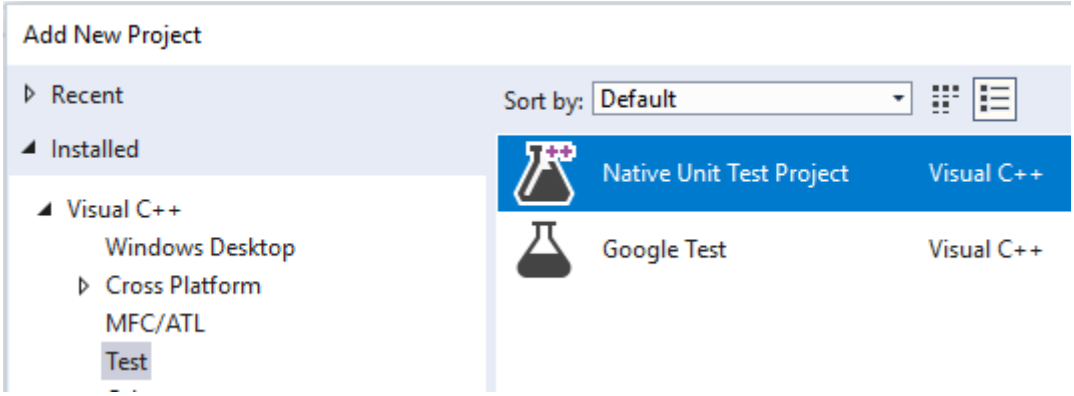
Создание проекта с набором тестов

Рассмотрим создание машинных unit-тестов на примере функции определения количества цифр в десятичной записи числа.

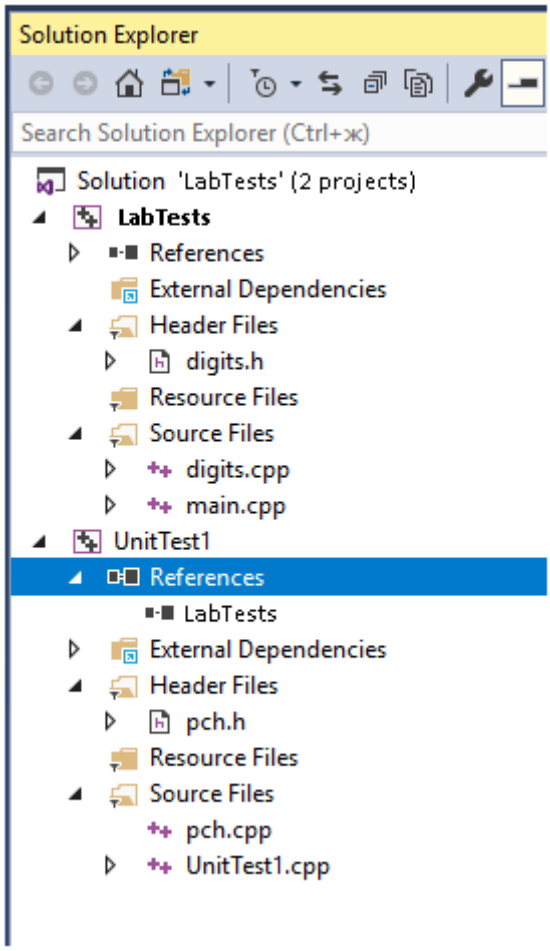
1. Создаем пустой проект, например, LabTests. Чтобы тесты располагались в одной папке с решением задачи, рекомендуется при создании проекта убрать признак «Поместить решение и проект в одном каталоге»
2. Добавляем в проект заголовочный файл digits.h
   `int digits(int n);`
3. Добавляем в проект файл digits.cpp. Чтобы показать, как работает разработка через тестирование, сделаем функцию с ошибками:

```
int digits(int n) {
  int k = 0;
  while (n > 0) {
       n /= 10;
  }
  return k;
}
```

4. Чтобы не получать сообщений об отсутствии точки входа, в проект добавляем исходный файл с пустым методом main.
5. Компилируем файл digits.cpp (Ctrl+F7), чтобы исправить синтаксические ошибки. Для компиляции установлена конфигурация **Debug** и платформа **x64**. Мы должны получить в папке проекта файл digits.obj. Он понадобится системе тестирования.
6. Добавляем в решение (Solution) еще один проект типа «Тесты». Выбираем тип проекта «Машинные тесты» (Native Unit Test Project)



7. Переходим в проект тестов.
8. Связываем тестовый проект с основным проектом. Пункт «Ссылки» (References) - «Добавить ссылку» (Add reference…)- находим проект LabTests и отмечаем его.



9. В проекте тестов находим файл исходных кодов unittest1.cpp . Он должен выглядеть так:

```
#include "stdafx.h"
#include "CppUnitTest.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Test1
{
	TEST_CLASS(UnitTest1)
	{
	public:

		TEST_METHOD(TestMethod1)
		{
			// TODO: Разместите здесь код своего теста
		}

	};
}
```

10. Для проверки работы тестирования можно создать тест, который всегда проходит. В тесте используются методы класса Assert. Если

соответствующий метод завершится успешно, тест считается пройденным.

```
TEST_METHOD(TestMethod1)
        {
            Assert::AreEqual(1, 1);
            Assert::AreNotEqual(1, 2);
            Assert::AreEqual(0.333, 1.0/3.0, 0.001);
            Assert::IsTrue(1 == 1);
        }
```

Для запуска тестов используем команду меню «Тест» - «Выполнить» - «Выполнить все тесты» (Test – Run – All Tests).

11. Подключить в этом файле заголовочный файл digits.h. При этом нужно указать путь к нему из папки тестов. При правильном наборе VS подсказывает доступные заголовочные файлы

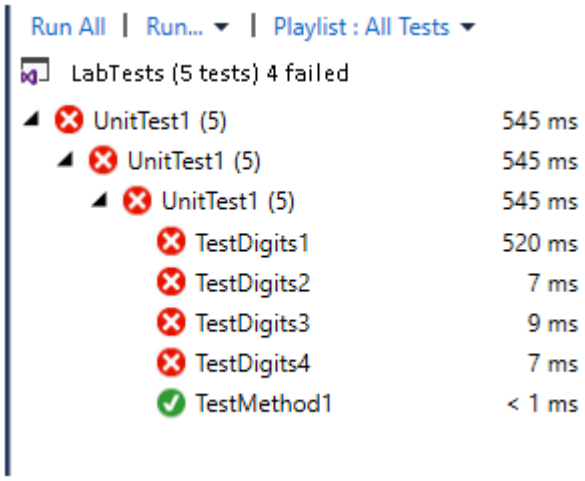
```
#include "..\ LabTests\digits.h"
```

12. Открыть свойства проекта (Properties) с тестами. Проверьте, что установлена Активная конфигурация Debug и платформа x64.
13. Добавить в свойствах проекта тестов в разделе «Компоновщик» - «Общие»- «Дополнительные каталоги библиотек» - «Изменить» (Linker – General – Additional Library Directories – Edit). Выбираем папку Debug тестируемого проекта (в ней должен был создаться файл digits.obj). Должен добавиться путь аналогичный следующему:
    D:\users\C++\ LabTests\ LabTests\Debug
14. Добавляем в свойствах «Компоновщик»-«Ввод»-«Дополнительные зависимости» - «Изменить» (Linker – Input- Additional Dependencies – Edit) имя файла digits.obj
15. Формируем набор тестов. Например такой:

```
TEST_METHOD(TestDigits1)
{
    Assert::AreEqual(digits(23), 2);
}
TEST_METHOD(TestDigits2)
{
    Assert::AreEqual(digits(5), 1);
}
TEST_METHOD(TestDigits3)
{
    Assert::AreEqual(digits(0), 1);
}
TEST_METHOD(TestDigits4)
{
    Assert::AreEqual(digits(-15), 2);
```

```
}
```

16. Допустимо как использовать отдельный тест для каждого случая проверки, так и объединять несколько проверок в одном тесте (как в примере теста, который всегда проходит).
17. Запускаем тестирование. Анализируем тесты, которые не проходят, вносим исправления в функцию digits и повторяем тестирование до тех пор, пока все тесты не пройдут.



Можно добавить другие тесты или использовать другие методы Assert, как это было показано в примере теста, который всегда проходит.

18. Если тестируемая функция меняет значения параметров, то тест может выглядеть так:

```
TEST_METHOD(TestChangeInt)
        {
// тесты для функции, вычеркивающей среднюю цифру в числе delMidlle
            int x = 123;
            delMidlle (x);
            Assert::AreEqual(x, 12);
            x = 1523;
            delMidlle (x);
            Assert::AreEqual(x, 123);
            x = 1;
            delMidlle (x);
            Assert::AreEqual(x, 0);

        }
```

19. При написании тестов к функциям, меняющим значения элементов массива, следует задавать массив до выполнения функции и массив после, задавая утверждение Assert в цикле для каждой пары значений.

Например, тест для функции сортировки buble_sort целочисленного массива:

```
TEST_METHOD(TestSort){
 int u_sort[]={7,3,-1,0,2};
 int sort[]={-1,0,2,3,7};
 buble_sort(u_sort,5);
 for (int i=0;i<5;++i)
  Assert::AreEqual(u_sort[i],sort[i]);
}
```

20. Тесты, проверяющие возникновение исключительной ситуации, формируются следующим образом

```
TEST_METHOD(TestExept) {
// создаем фрагмент кода, в котором должно возникнуть исключение
// например, деление на 0  в функции hyperbola, тип исключения - int
      auto func = [] {double y= hyperbola(0.0);};
      Assert::ExpectException<int>(func);
}
```

Если с написанием функции, вызывающей исключение, возникают проблемы можно оформить тест следующим образом:

```
TEST_METHOD(TestExept) {
      try {
            Assert::AreEqual(0.0, hyperbola(0.0));
            Assert::Fail("ERROR"); //ошибка, если попали сюда
      }
      catch(int a){
            Assert::IsTrue(true); //OK
      }
 }
```

Более подробная информация по использованию машинных unit-тестов в VS:

Справочник по API Microsoft.VisualStudio.TestTools.CppUnitTestFramework

https://docs.microsoft.com/ru-ru/visualstudio/test/microsoft-visualstudio-testtools-cppunittestframework-api-reference?view=vs-2019