**<https://www.youtube.com/watch?v=jyQcCWWcIQU>**

**огромный yourtube по тестированию**

**Ищи здесь**

**https**://www.bogotobogo.com/cplusplus/google\_unit\_test\_gtest.php

**Получите платформу тестирования Google C++**

1. Загрузите последнюю [версию фреймворка](http://code.google.com/p/googletest/downloads/list) gtest
2. Распаковать в C:\gtest

**Создание библиотек фреймворка**

1. Открыть C:\gtest\msvc\gtest.sln в Visual Studio
2. Установите конфигурацию на "Debug"
3. Решение для сборки

**Создание и настройка Тестового проекта**

1. Создайте новое решение и выберите шаблон Visual C++ > Win32 > Консольное приложение Win32
2. Щелкните правой кнопкой мыши вновь созданный проект и выберите пункт Свойства
3. Измените конфигурацию для отладки.
4. Свойства конфигурации > C/C++ > Общие > Дополнительные каталоги включения: Добавить C:\gtest\include
5. Свойства конфигурации > C/C++ > Генерация кода > Библиотека среды выполнения: Если ваш код ссылается на среду выполнения DLL, выберите Многопоточную отладку DLL (/MDd). Если нет, выберите Многопоточную отладку (/MTd).
6. Свойства конфигурации > Компоновщик > Общие > Дополнительные каталоги библиотек: Добавьте C:\gtest\msvc\gtest\Debug или C:\gtest\msvc\gtest-md\Debug , в зависимости от расположения gtestd.lib
7. Свойства конфигурации > Компоновщик > Ввод > Дополнительные зависимости: Добавить gtestd.lib

**Проверка Того, Что Все Работает**

1. Откройте cpp в тестовом проекте, содержащем функцию main() .
2. Вставьте следующий код:
3. #include "stdafx.h"
4. #include <iostream>
5. #include "gtest/gtest.h"
6. TEST(sample\_test\_case, sample\_test)
7. {
8. EXPECT\_EQ(1, 1);
9. }
10. int main(int argc, char\*\* argv)
11. {
12. testing::InitGoogleTest(&argc, argv);
13. RUN\_ALL\_TESTS();
14. std::getchar(); // keep console window open until Return keystroke

}

1. Отладка > Начать отладку

Если все сработало, вы должны увидеть, как появится окно консоли с результатами модульного тестирования.

<https://www-badprog-com.translate.goog/c-google-test-setup-visual-studio-by-hand?_x_tr_sl=en&_x_tr_tl=ru&_x_tr_hl=ru&_x_tr_pto=nui,sc>

**C ++ - Google Test - Настройка Visual Studio вручную**

Отправлено Mi-K в четверг, 3 августа 2017 г., в 18:19

Прежде чем играть с библиотекой Google Test, необходимо установить ее.

Это то, из чего состоит это руководство, набор элементов для его настройки.

В нашем примере мы собираемся использовать Visual Studio 2017, то есть в Windows, без интерфейса командной строки (CLI) и без макроса RUN\_ALL\_TESTS ().

**Прежде всего**

Для реализации этого руководства по тестированию Google необходимы два инструмента:

* **Visual Studio**
* **Google Test**

**Visual Studio**

Допустим, вы уже скачали **Visual Studio 2017** , но если нет, вы всегда можете это сделать.

В нашем случае это **версия сообщества** :

* **Сообщество Visual Studio 2017** :  [https://www.visualstudio.com/downloads/](https://translate.google.com/website?sl=en&tl=ru&nui=1&prev=search&u=https://www.visualstudio.com/downloads/)

**Google Test**

Конечно, нам понадобится фреймворк Google Test.

В нашей команде мы будем использовать **версию 1.8.0** .

Чтобы узнать точно, какую версию вы используете, я рекомендую использовать официальную веб-страницу Google Test на GitHub и выбрать нужную версию (в нашем случае выпуск 1.8.0):

* **Google Test 1.8.0** :  [https://github.com/google/googletest/releases](https://translate.google.com/website?sl=en&tl=ru&nui=1&prev=search&u=https://github.com/google/googletest/releases)

Для нашего руководства я выбираю установку Google Test в следующий каталог:

* **C: \ dev \ c ++ \ mylib \ googletest-release-1.8.0**

**Проекты**

Из Visual Studio мы собираемся создать **решение** с **3 разными проектами** .

Итак, в окне Новый проект у нас будет для первого проекта:

* Имя: **BadprogProject**
* Расположение:  **C: \ dev \ c ++ \ vs \**
* Название **решения** :  **Solution\_1**

Для второго проекта:

* Имя:  **BadprogProjectTests**
* Расположение:  **C: \ dev \ c ++ \ vs \ Solution\_1**

Для третьего проекта:

* Имя: **GoogleTestFramework**
* Расположение:  **C: \ dev \ c ++ \ vs \ Solution\_1**

Таким образом решение называется **Solution\_1,** и три проекта:

1. **BadprogProject**
2. **BadprogProjectTests**
3. **GoogleTestFramework**

Как видите, первый - классический проект, второй - проект тестирования, а последний - фреймворк Google Test.

Все проекты будут **без** в **предкомпилированном заголовке** опции.

Причина в том, что с этой опцией код не будет работать должным образом, MSVC запрашивает некоторые файлы, такие как **stdafx.h,**  даже во внешних библиотеках, таких как Google Test.

И мы этого не хотим, мы сняли флажок с опции Предварительно скомпилированный заголовок в каждом проекте.

**BadprogProject**

Проект является этим **консольным приложением Win32** с дополнительными **параметрами**  без предварительно **скомпилированного заголовка** (не отмечены).

**BadprogProjectTests**

Этот проект является **консольным приложением Win32** с дополнительными **включеннымибез**  предварительного **скомпилированного заголовка**  (не отмечен ).

**GoogleTestFramework**

Проект является этот  **консольным приложением Win32**  с **типом** Приложения, ТАКИМ как  **статическая Библиотека,** и с дополнительными **параметрами** без предварительно  **скомпилированного заголовка** (не отмечен).

**Макросы**

**Создание макросов**

У нас есть 3 проекта, использующих одну и ту же библиотеку (Google Test), мы создадим 2 макроса: один для домашней страницы Google Test, а другой - для их включения.

В Visual Studio> Просмотр> Другие окна> Диспетчер свойств.

Теперь вы должны увидеть 3 проекта с 4 разными папками:

* **Отладка | Win32**
* **Отладка | x64**
* **Релиз | Win32**
* **Релиз | x64**

Для этого урока мы будем использовать только **Release |**Версия **Win32** , так что вы можете использовать другие, если хотите.

Из выбранного проекта ( например, **GoogleTestFramework** ).

Щелкните белый треугольник слева от Release | Win32 для отображения всех параметров.

Внутри вы должны увидеть что-то вроде этого:

* **Microsoft.Cpp.Win32.user**
* **Оптимизация всей программы**
* **заявка**
* **Поддержка Unicode**
* **Базовые библиотеки Windows**

Итак, возьмем первый в списке ( **Microsoft.Cpp.Win32.user** ) и **щелкните его правой кнопкой мыши> Свойства** .

Появились соответствующие **страницы свойств** .

С помощью маклева **Общие свойства> Пользовательские запросы** .

Теперь на правой панели вы нажимаете кнопку « **Добавить макрос»** .

Появилось окно « **Добавить макрос пользователя** ».

Необходимо заполнить два текстовых поля:

* Имя: **GOOGLE\_TEST\_HOME**
* Значение:  **C: \ dev \ c ++ \ mylib \ googletest-release-1.8.0 \ googletest**

**Установите этот**флажок **как переменную среду в среде сборки** .

Щелкните ОК.

Этот макрос теперь добавлен в список макросов.

Давайте добавим еще один для папки:

* Имя:  **GOOGLE\_TEST\_INCLUDE**
* Значение: **C: \ dev \ c ++ \ mylib \ googletest-release-1.8.0 \ googletest \ include**

Еще раз отметьте опцию  **Установите этот макрос как переменную среды в среде сборки** .

Теперь 2 макроса готовы, не нажимайте **Применить> ОК** .

**Использование макросов**

Вернемся к нашим проектам, **выбрав Visual Studio> Вид> Обозреватель решений** .

Мы собираемся установить каждый проект с новыми макросами.

Итак, щелкните правой кнопкой **мыши** проект **BadprogProject** ,: « **Свойства»> «Свойства конфигурации»> «C / C ++»> «Общие»** .

*Если вы не видите подменю C / C ++ в свойствах конфигурации, вы можете вместо этого использовать****каталоги VC ++> Включить каталоги****.*

Справа находится параметр **Дополнительные**каталоги **включения** .

Щелкните по нему, и справа вы увидите раскрывающееся меню с уникальной опцией:  **<Изменить ...>** .

Щелкните по нему.

Появились каталоги Include.

В правом нижнем углу у вас есть кнопка « **Макросы >>»** , нажмите ее.

Он должен открыть список текущих макросов, используемых в вашей среде Visual Studion.

Вы можете проверить наличие наших новых макросов, поискав по букве G, чтобы увидеть их.

К сожалению, напрямую скопировать их слева невозможно.

Это было просто для того, чтобы убедиться, что наши макросы сохранены.

ОК, с левой стороны щелкните значок крошечной папки, чтобы новый каталог включить, и введите:

$ (GOOGLE\_TEST\_HOME)

Затем еще раз введите:

$ (GOOGLE\_TEST\_INCLUDE)

Не забудьте скобки, иначе ничего не получится.

На панели **вычисленных значений** .

Теперь у нас установлено 2 подключаемых каталога.

Щелкните **ОК,** чтобы закончить.

Затем **Применить> ОК** .

Сделайте то же самое для BadprogProjecTests.

А для GoogleTestFramework вместо использования свойств C / C ++ используйте следующее:

Щелкните правой кнопкой мыши GoogleTestFramework> **Свойства> Каталоги VC ++** > справа **включить** каталоги и добавить макросы.

**Добавление файлов**

Мы должны добавить несколько файлов в 3 проекта.

**BadprogProject**

Щелкните правой кнопкой мыши> **Добавить> Новый элемент ...> Файл заголовка (.h)> Calculation.h** (как имя) **> Добавить** .

Внутри введите следующий код:

// BadproG.com

#ifndef BADPROG\_PROJECT\_CALCULATION\_H\_

#define BADPROG\_PROJECT\_CALCULATION\_H\_

/ \*\*

\* Добавление первого ко второму.

\* /

int add (сначала const int, затем const int) {

возврат (первый + второй);

}

/ \*\*

\* Вычитание первого варианта из второго.

\* /

int substract (сначала const int, затем const int) {

возврат (первый - второй);

}

/ \*\*

\* Умножение первого на второй.

\* /

int multiply (сначала const int, затем const int) {

возврат (первый \* второй);

}

/ \*\*

\* Деление первого варианта на второй.

\* /

int div (сначала const int, затем const int) {

возврат (первый / второй);

}

#endif

**BadprogProjectTests**

Удалите все и добавить следующий фрагмент кода в BadprogProjectTests.cpp:

// BadproG.com

#include "gtest / gtest.h"

#include "../BadprogProject/Calculation.h"

// добавляем

ТЕСТ (Test\_Calculation, Добавление)

{

EXPECT\_EQ (5, добавить (2, 3));

EXPECT\_EQ (-7; добавить (-2; -3));

}

// вычитание

ТЕСТ (Тест\_расчет, вычитание)

{

EXPECT\_EQ (-1; вычесть (2, 3));

EXPECT\_EQ (-7, вычесть (-2, -3));

}

// умножение

ТЕСТ (Test\_Calculation, Умножение)

{

EXPECT\_EQ (6, умножить (2, 3));

EXPECT\_EQ (-7; умножить (-2; -3));

}

// деление

ТЕСТ (Test\_Calculation, Деление)

{

EXPECT\_EQ (2, разделить (10, 5));

EXPECT\_EQ (-7, разделить (15, -3));

}

**GoogleTestFramework**

Щелкните правой кнопкой мыши GoogleTestFramework> **Добавить> Существующий элемент ...** > выберите файл:

* **C: \ dev \ c ++ \ mylib \ googletest-release-1.8.0 \ googletest \ include \ gtest \ gtest.h.**

Сделайте то же самое для следующих файлов:

* **C: \ dev \ c ++ \ mylib \ googletest-release-1.8.0 \ googletest \ src \ gtest\_main.cc**
* **C: \ dev \ c ++ \ mylib \ googletest-release-1.8.0 \ googletest \ src \ gtest-all.cc**

Добавляя **gtest\_main.cc** в наш проект, мы пропускаем использование **макроса RUN\_ALL\_TESTS ()** и то же самое для функций  **:: testing :: InitGoogleTest (& argc, argv)** .

**Создание решения**

Перед построением решения мы должны:

1. **Установка BadprogProjectTests в качестве запускаемого проекта.**
2. **Добавление GoogleTestFramework в качестве ссылки.**

1. Щелкните правой кнопкой мыши BadprogProjectTests **> Установить как запускаемый проект** .

2. В BadprogProjectTests правой кнопкой мыши подменю «Ссылки»> « **Добавить»ссылку ... »** >« Выберите **GoogleTestFramework »>« ОК »** .

**Создание решения**

Пришло время перестроить решение> правой кнопкой мыши Solution\_1> **Восстановить решение** .

Все должно работать, и в выходных данных появится следующее сообщение:

1> ------ Перестроить Все запущено: Проект: GoogleTestFramework, Конфигурация: Отладка Win32 ------

2> ------ Перестроить Все запущено: Проект: BadprogProject, Конфигурация: Отладка Win32 ------

2> stdafx.cpp

1> gtest\_main.cc

2> BadprogProject.cpp

2> Создание кода ...

2> BadprogProject.vcxproj -> C: \ dev \ c ++ \ vs \ Solution\_1 \ Debug \ BadprogProject.exe

2> BadprogProject.vcxproj -> C: \ dev \ c ++ \ vs \ Solution\_1 \ Debug \ BadprogProject.pdb (частичный PDB)

1> gtest-all.cc

1> Создание кода ...

1> GoogleTestFramework.vcxproj -> C: \ dev \ c ++ \ vs \ Solution\_1 \ Debug \ GoogleTestFramework.lib

3> ------ Перестроить Все запущено: Проект: BadprogProjectTests, Конфигурация: Отладка Win32 ------

3> stdafx.cpp

3> BadprogProjectTests.cpp

3> Создание кода ...

3> BadprogProjectTests.vcxproj -> C: \ dev \ c ++ \ vs \ Solution\_1 \ Debug \ BadprogProjectTests.exe

3> BadprogProjectTests.vcxproj -> C: \ dev \ c ++ \ vs \ Solution\_1 \ Debug \ BadprogProjectTests.pdb (частичный PDB)

========== Перестроить все: 3 успешно, 0 неудачно, 0 пропущено ==========

**Тестирование наших тестов**

Вот и мы, пора посмотреть, вся ли наша работа была полезной.

В Visual Studio> **Отладка> Начать без отладки** .

Cmd.exe должен появиться со всеми тестами RUN (зеленый) и FAILED (красный).

И следующий дисплей:

Запуск main () из gtest\_main.cc

[==========] Запуск 4 теста из 1 теста.

[----------] Настройка глобальной тестовой среды.

[----------] 4 теста из Test\_Calculation

[RUN] Test\_Calculation.Adding

c: \ dev \ c ++ \ vs \ solution\_1 \ badprogprojecttests \ badprogprojecttests.cpp (10): ошибка: ожидается: -7

Чтобы быть равным: добавить (-2, -3)

Что: -5

[FAILED] Test\_Calculation.Adding (1 ms)

[RUN] Test\_Calculation.Substracting

c: \ dev \ c ++ \ vs \ solution\_1 \ badprogprojecttests \ badprogprojecttests.cpp (17): ошибка: ожидается: -7

Чтобы быть равным: умножить (-2, -3)

Что: 6

[FAILED] Test\_Calculation.Substracting (1 ms)

[RUN] Test\_Calculation.Multiplying

c: \ dev \ c ++ \ vs \ solution\_1 \ badprogprojecttests \ badprogprojecttests.cpp (24): ошибка: ожидается: -7

Чтобы быть равным: умножить (-2, -3)

Что: 6

[FAILED] Test\_Calculation. Умножение (1 мс)

[RUN] Test\_Calculation. Деление

c: \ dev \ c ++ \ vs \ solution\_1 \ badprogprojecttests \ badprogprojecttests.cpp (31): ошибка: ожидается: -7

Чтобы быть равным: добавить (15, -3)

Что: 12

[FAILED] Test\_Calculation.Diving (0 мс)

[----------] 4 теста из Test\_Calculation (всего 4 мс)

[----------] Разборка глобальной тестовой среды

[==========] Выполнено 4 теста из 1 теста. (Всего 5 мс)

[ПРОЙДЕНО] 0 тестов.

[НЕ ПРОЙДЕНО] 4 теста, перечисленных ниже:

[FAILED] Test\_Calculation.Adding

[FAILED] Test\_Calculation.Substracting

[FAILED] Test\_Calculation.Multiplying

[FAILED] Test\_Calculation.Diving

4 НЕПРАВИЛЬНЫХ ИСПЫТАНИЯ

Нажмите любую инструкцию для продолжения. . .

Наши НЕУДАЧИ - нормальные.

Действительно, в некоторых наших тестах мы передали неверные аргументы, чтобы проверить, хорошо ли работают тесты.

**Заключение**

Интересный способ узнать больше о том, как модульное тестирование работает с Google Test и Visual Studio.

Не стесняйтесь и адаптируйте свои тесты в соответствии с вашими потребностями.

Поздравляем, если вы дочитали до этого места, у вас все получилось, отличная работа! смех

[‹C ++ - Google Test - Настройка проектов для использования библиотеки gtest](https://www-badprog-com.translate.goog/c-google-test-set-up-projects-to-use-the-gtest-library?_x_tr_sl=en&_x_tr_tl=ru&_x_tr_hl=ru&_x_tr_pto=nui,sc)[вверх](https://www-badprog-com.translate.goog/c-google-test?_x_tr_sl=en&_x_tr_tl=ru&_x_tr_hl=ru&_x_tr_pto=nui,sc)[C ++ - Google Test - Использование Makefile для создания тестов ›](https://www-badprog-com.translate.goog/c-google-test-using-a-makefile-to-generate-tests?_x_tr_sl=en&_x_tr_tl=ru&_x_tr_hl=ru&_x_tr_pto=nui,sc)

<https://chromium.googlesource.com/external/github.com/google/googletest/+/refs/tags/release-1.8.0/googletest/docs/Primer.md>

И еще много отсылок

Introduction: Why Google C++ Testing Framework?

*Google C++ Testing Framework* helps you write better C++ tests.

No matter whether you work on Linux, Windows, or a Mac, if you write C++ code, Google Test can help you.

So what makes a good test, and how does Google C++ Testing Framework fit in? We believe:

1. Tests should be *independent* and *repeatable*. It's a pain to debug a test that succeeds or fails as a result of other tests. Google C++ Testing Framework isolates the tests by running each of them on a different object. When a test fails, Google C++ Testing Framework allows you to run it in isolation for quick debugging.
2. Tests should be well *organized* and reflect the structure of the tested code. Google C++ Testing Framework groups related tests into test cases that can share data and subroutines. This common pattern is easy to recognize and makes tests easy to maintain. Such consistency is especially helpful when people switch projects and start to work on a new code base.
3. Tests should be *portable* and *reusable*. The open-source community has a lot of code that is platform-neutral, its tests should also be platform-neutral. Google C++ Testing Framework works on different OSes, with different compilers (gcc, MSVC, and others), with or without exceptions, so Google C++ Testing Framework tests can easily work with a variety of configurations. (Note that the current release only contains build scripts for Linux - we are actively working on scripts for other platforms.)
4. When tests fail, they should provide as much *information* about the problem as possible. Google C++ Testing Framework doesn't stop at the first test failure. Instead, it only stops the current test and continues with the next. You can also set up tests that report non-fatal failures after which the current test continues. Thus, you can detect and fix multiple bugs in a single run-edit-compile cycle.
5. The testing framework should liberate test writers from housekeeping chores and let them focus on the test *content*. Google C++ Testing Framework automatically keeps track of all tests defined, and doesn't require the user to enumerate them in order to run them.
6. Tests should be *fast*. With Google C++ Testing Framework, you can reuse shared resources across tests and pay for the set-up/tear-down only once, without making tests depend on each other.

Since Google C++ Testing Framework is based on the popular xUnit architecture, you‘ll feel right at home if you’ve used JUnit or PyUnit before. If not, it will take you about 10 minutes to learn the basics and get started. So let's go!

*Note:* We sometimes refer to Google C++ Testing Framework informally as *Google Test*.

Setting up a New Test Project

To write a test program using Google Test, you need to compile Google Test into a library and link your test with it. We provide build files for some popular build systems: msvc/ for Visual Studio, xcode/ for Mac Xcode, make/ for GNU make, codegear/ for Borland C++ Builder, and the autotools script (deprecated) and CMakeLists.txt for CMake (recommended) in the Google Test root directory. If your build system is not on this list, you can take a look at make/Makefile to learn how Google Test should be compiled (basically you want to compile src/gtest-all.cc with GTEST\_ROOT and GTEST\_ROOT/include in the header search path, where GTEST\_ROOT is the Google Test root directory).

Once you are able to compile the Google Test library, you should create a project or build target for your test program. Make sure you have GTEST\_ROOT/include in the header search path so that the compiler can find "gtest/gtest.h" when compiling your test. Set up your test project to link with the Google Test library (for example, in Visual Studio, this is done by adding a dependency on gtest.vcproj).

If you still have questions, take a look at how Google Test's own tests are built and use them as examples.

Basic Concepts

When using Google Test, you start by writing *assertions*, which are statements that check whether a condition is true. An assertion's result can be *success*, *nonfatal failure*, or *fatal failure*. If a fatal failure occurs, it aborts the current function; otherwise the program continues normally.

*Tests* use assertions to verify the tested code's behavior. If a test crashes or has a failed assertion, then it *fails*; otherwise it *succeeds*.

A *test case* contains one or many tests. You should group your tests into test cases that reflect the structure of the tested code. When multiple tests in a test case need to share common objects and subroutines, you can put them into a *test fixture* class.

A *test program* can contain multiple test cases.

We'll now explain how to write a test program, starting at the individual assertion level and building up to tests and test cases.

Assertions

Google Test assertions are macros that resemble function calls. You test a class or function by making assertions about its behavior. When an assertion fails, Google Test prints the assertion‘s source file and line number location, along with a failure message. You may also supply a custom failure message which will be appended to Google Test’s message.

The assertions come in pairs that test the same thing but have different effects on the current function. ASSERT\_\* versions generate fatal failures when they fail, and **abort the current function**. EXPECT\_\* versions generate nonfatal failures, which don‘t abort the current function. Usually EXPECT\_\* are preferred, as they allow more than one failures to be reported in a test. However, you should use ASSERT\_\* if it doesn’t make sense to continue when the assertion in question fails.

Since a failed ASSERT\_\* returns from the current function immediately, possibly skipping clean-up code that comes after it, it may cause a space leak. Depending on the nature of the leak, it may or may not be worth fixing - so keep this in mind if you get a heap checker error in addition to assertion errors.

To provide a custom failure message, simply stream it into the macro using the << operator, or a sequence of such operators. An example:

ASSERT\_EQ(x.size(), y.size()) << "Vectors x and y are of unequal length";

for (int i = 0; i < x.size(); ++i) {

EXPECT\_EQ(x[i], y[i]) << "Vectors x and y differ at index " << i;

}

Anything that can be streamed to an ostream can be streamed to an assertion macro--in particular, C strings and string objects. If a wide string (wchar\_t\*, TCHAR\* in UNICODE mode on Windows, or std::wstring) is streamed to an assertion, it will be translated to UTF-8 when printed.

Basic Assertions

These assertions do basic true/false condition testing.

| **Fatal assertion** | **Nonfatal assertion** | **Verifies** |
| --- | --- | --- |
| ASSERT\_TRUE(*condition*); | EXPECT\_TRUE(*condition*); | *condition* is true |
| ASSERT\_FALSE(*condition*); | EXPECT\_FALSE(*condition*); | *condition* is false |

Remember, when they fail, ASSERT\_\* yields a fatal failure and returns from the current function, while EXPECT\_\* yields a nonfatal failure, allowing the function to continue running. In either case, an assertion failure means its containing test fails.

*Availability*: Linux, Windows, Mac.

Binary Comparison

This section describes assertions that compare two values.

| **Fatal assertion** | **Nonfatal assertion** | **Verifies** |
| --- | --- | --- |
| ASSERT\_EQ(*val1*,*val2*); | EXPECT\_EQ(*val1*,*val2*); | *val1* == *val2* |
| ASSERT\_NE(*val1*,*val2*); | EXPECT\_NE(*val1*,*val2*); | *val1* != *val2* |
| ASSERT\_LT(*val1*,*val2*); | EXPECT\_LT(*val1*,*val2*); | *val1* < *val2* |
| ASSERT\_LE(*val1*,*val2*); | EXPECT\_LE(*val1*,*val2*); | *val1* <= *val2* |
| ASSERT\_GT(*val1*,*val2*); | EXPECT\_GT(*val1*,*val2*); | *val1* > *val2* |
| ASSERT\_GE(*val1*,*val2*); | EXPECT\_GE(*val1*,*val2*); | *val1* >= *val2* |