gtest提供了多种事件机制，非常方便我们在案例之前或之后做一些操作。总结一下gtest的事件一共有3种：

1. 全局的，所有案例执行前后。

2. TestSuite，在某一批案例中第一个案例前，最后一个案例执行后。

3. TestCase，每个TestCase前后。

一、全局事件

1. SetUp()方法在所有案例执行前执行

2. TearDown()方法在所有案例执行后执行

class FooEnvironment : public testing::Environment

{

public:

    virtual void SetUp()

    {

        std::cout << "Foo FooEnvironment SetUP" << std::endl;

    }

    virtual void TearDown()

    {

        std::cout << "Foo FooEnvironment TearDown" << std::endl;

    }

};

还需要告诉gtest添加这个全局事件，我们需要在main函数中通过testing::AddGlobalTestEnvironment方法将事件挂进来，也就是说，我们可以写很多个这样的类，然后将他们的事件都挂上去。

int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    testing::AddGlobalTestEnvironment(new FooEnvironment);  
    testing::InitGoogleTest(&argc, argv);  
    return RUN\_ALL\_TESTS();  
}

TestSuite事件

我们需要写一个类，继承testing::Test，然后实现两个静态方法

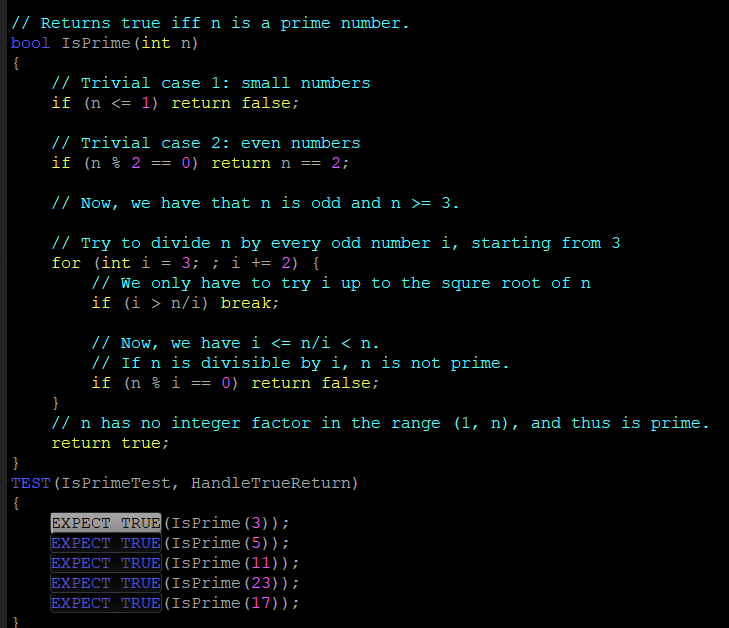
1. SetUpTestCase() 方法在第一个TestCase之前执行

2. TearDownTestCase() 方法在最后一个TestCase之后执行

参数化

在设计测试案例时，经常需要考虑给被测函数传入不同的值的情况，通常是写一个通用方法，然后编写在测试案例调用它，但是这样会有很多重复性，鉴于这个问题吗，提供了一个灵活的参数化测试方案。

原本方案：



假如要编写判断结果为TRUE 的测试案例，需要传入一系列数值让函数IsPrime去判断是否为True（当然，即使传入再多值也无法确保函数正确，呵呵），因此我需要这样编写如下的测试案例：如上。

参数化后的方案：

1. 告诉gtest 参数类型是什么

必须要先添加一个类。继承自 testing::TestWitnParam<T>，其中T 是需要参数化的参数类型，比如上述案例，需要参数化一个int 类型的参数。

class IsPrimeParamTest : public::testing::TestWithParam<int>

{

};

1. 告诉gtest 拿到参数的值后，具体做什么样的测试

此时，需要使用一个新的宏，TEST\_P。关于这个P ，即：parameterized：参数。在TEST\_P里，使用GetParam()获取当前的参数的具体值。

类型参数化

Gtest提供了应付各种不同类型的数据时的方案，以及参数化类型的方案

首先要了解一下类型化测试。

先定义一个模板类，继承testing::Test：

template <typename T>  
class FooTest : public testing::Test {  
 public:  
  IMG_256  
  typedef std::list<T> List;  
  static T shared\_;  
  T value\_;  
};

然后定义需要测试到的具体数据类型，比如下面定义了需要测试的char,int ,unsigned int;

typedef testing::Types<char, int, unsigned int> MyTypes;  
TYPED\_TEST\_CASE(FooTest, MyTypes);

在声明模板的数据类型时，使用TypeParam

TYPED\_TEST(FooTest, DoesBlah) {

// Inside a test, refer to the special name TypeParam to get the type

// parameter. Since we are inside a derived class template, C++ requires

// us to visit the members of FooTest via 'this'.

TypeParam n = this->value\_;

// To visit static members of the fixture, add the 'TestFixture::'

// prefix.

n += TestFixture::shared\_;

// To refer to typedefs in the fixture, add the 'typename TestFixture::'

// prefix. The 'typename' is required to satisfy the compiler.

typename TestFixture::List values;

values.push\_back(n);

}

上述例子不够灵活，需要事先知道类型的列表，Gtest还提供一种更加灵活的类型参数化的方式，允许在完成测试的逻辑代码之后再去考虑需要参数化的类型列表，ingIE可以重复的使用这个类型列表。如下

**死亡测试**

一、是指：程序的崩溃。通常测试过程中，我们需需要考虑各种输入，有的输入可能直接导致程序崩溃，这时我们需要检查程序是否按照预期的方式挂掉。Gtest的死亡测试能做到在一个安全的环境下执行崩溃的测试案例，同时又对崩溃结果进行验证。

二、使用的宏

|  |  |  |
| --- | --- | --- |
| Fatal assertion（致命的断言） | Nonfatal assertion | Verifies（证明） |
| ASSERT\_DEATH(statement, regex`); | EXPECT\_DEATH(statement, regex`); | statement crashes with the given error |
| ASSERT\_EXIT(statement, predicate, regex`); | EXPECT\_EXIT(statement, predicate, regex`); | statement exits with the given error and its exit code matches predicate |

语句因给定错误而崩溃

语句以给定的错误退出，其退出代码与谓词匹配

由于有些异常只在Debug下抛出，因此还提供了\*\_DEBUG\_DEATH，用来处理Debug和Realease下的不同。

1. \*\_DEATH(statement ,regex)
2. stetement 是被测试的代码语句
3. Regex 是正则表达式，用来匹配异常时在stderr中输出的内容

如下

void Foo()  
{  
    int \*pInt = 0;  
    \*pInt = 42 ;  
}  
  
TEST(FooDeathTest, Demo)  
{  
    EXPECT\_DEATH(Foo(), "");  
}