

# 山东大学

SHANDONG UNIVERSITY



C++ Test

实验报告

课程名称：软件质量保证与测试技术

姓名：武敬信

学号：202000800525

专业班级：21软件工程1班

授课教师：康钦马

2024 年 5 月 23 日

- 
- 1 实验目的
  - 2 实验环境
  - 3 实验步骤
    - 3.1 安装 C++ Test
    - 3.2 C++ Test 快速测试
    - 3.3 静态测试
    - 3.4 动态测试
    - 3.5 生成报表
    - 3.6 C++ Test 高级功能
    - 3.7 测试用例分析
    - 3.8 调试测试用例
    - 3.9 Data Source
    - 3.10 桩函数设置
    - 3.11 导入导出测试用例
    - 3.12 Test Objects
    - 3.13 覆盖率分析
    - 3.14 回归测试
    - 3.15 RuleWizard 定制规则

## 1 实验目的

C++Test 是一个 C/C++单元测试工具，自动测试任何 C/C++类、函数或部件，而不需要编写一个测试用例、测试驱动程序或桩调用。通过实验需要掌握 C++ Test 的集中测试功能，并了解一些高级使用的方法。

## 2 实验环境

虚拟机：Windows XP

软件：C++ Test V6.0.0.5

CPU：AMD Ryzen 7 5800H

内存：8GB

## 3 实验步骤

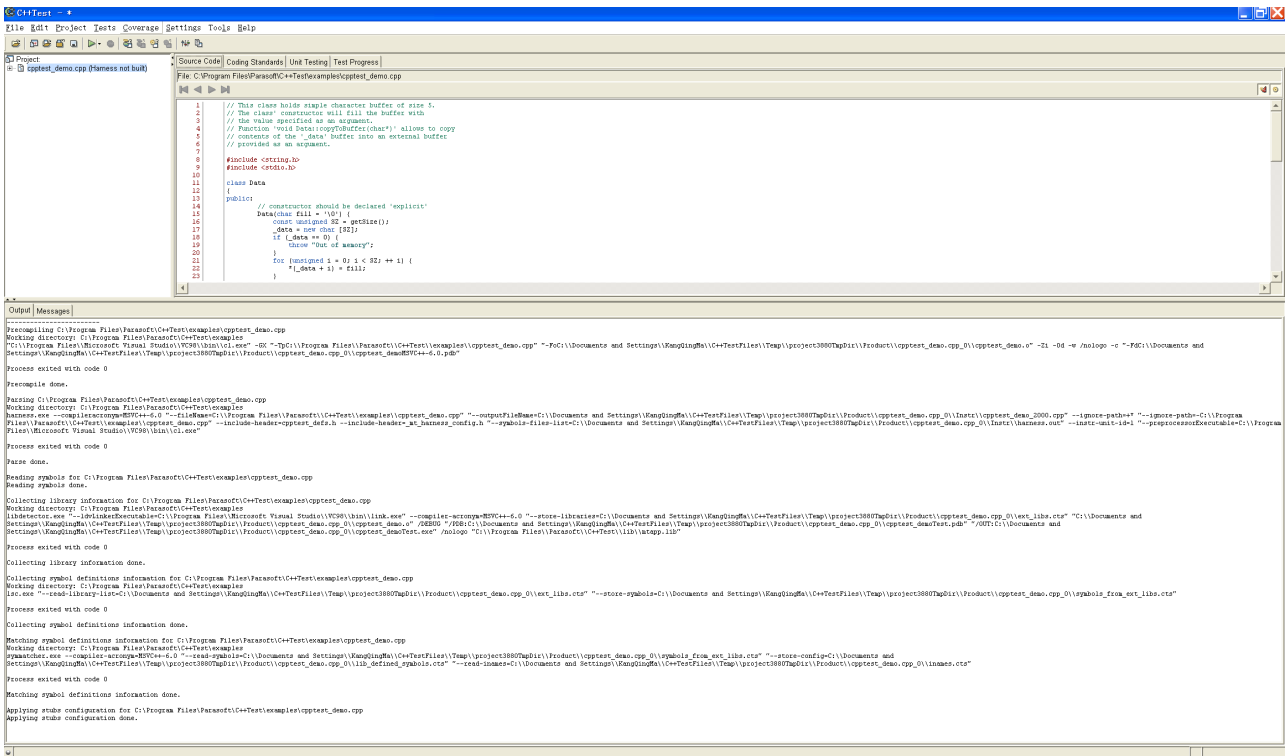
### 3.1 安装 C++ Test

安装 C++ Test，将日期设置为 2005 年。

### 3.2 C++ Test 快速测试

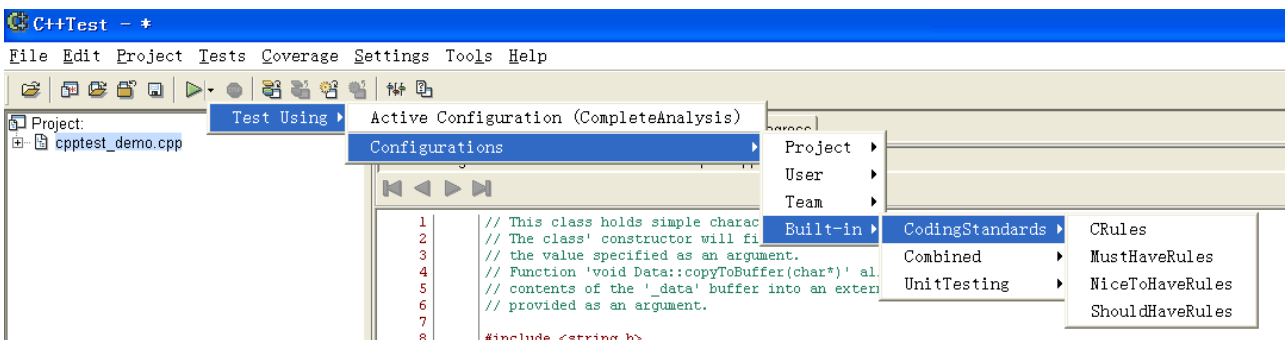
打开 C++ Test，选择 `file/open file`，选择 C++Test 安装目录下 `examples/cppptest_demo.cpp`，这个 cpp 文件将出现在当前的project 下。

在当前 project 下，右击 `cppptest_demo.cpp`，选择 `read symbols`，此时 C++ Test 将剖析这个源程序，分析出此文件的文件结构。



### 3.3 静态测试

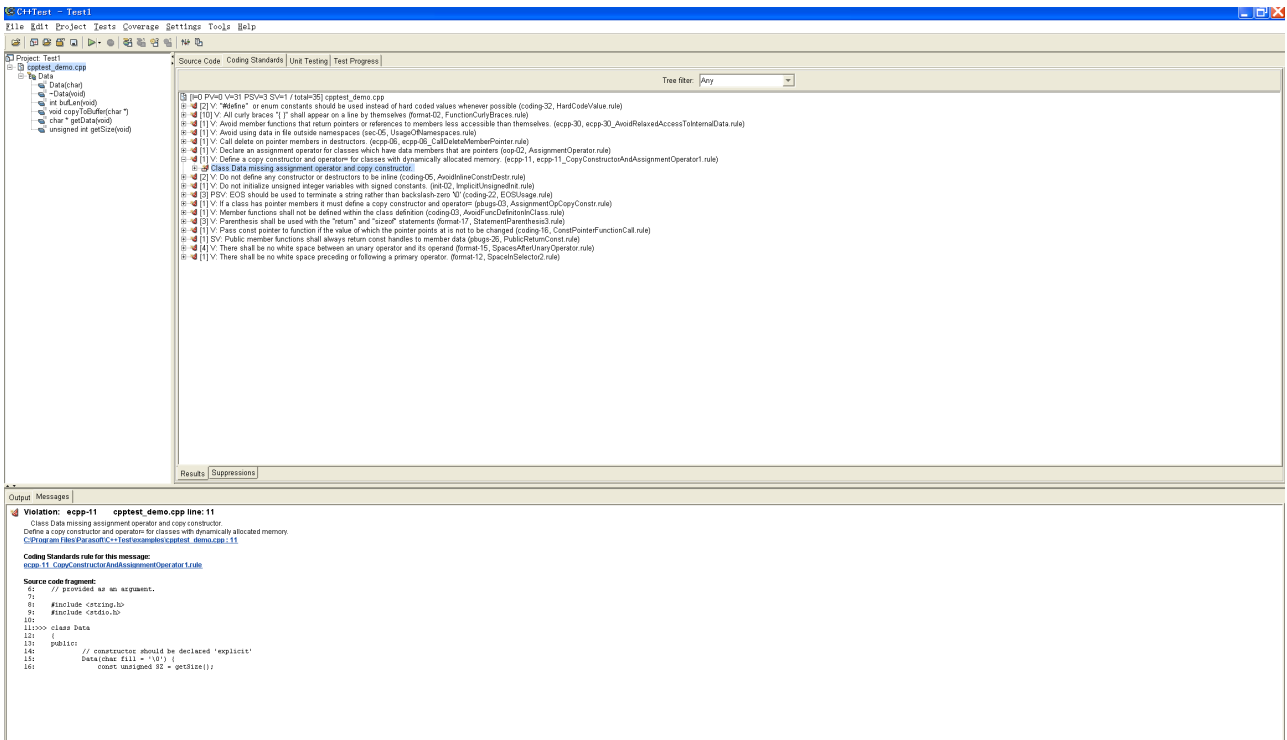
在向右三角型旁边的下拉箭头，选择内置的编码规则项目。



C++ Test 将自动完成对源代码的静态测试，也就是我们所说的代码走查，走查所用到的规范可以在静态测试标签的 **rule manage** 下看到。

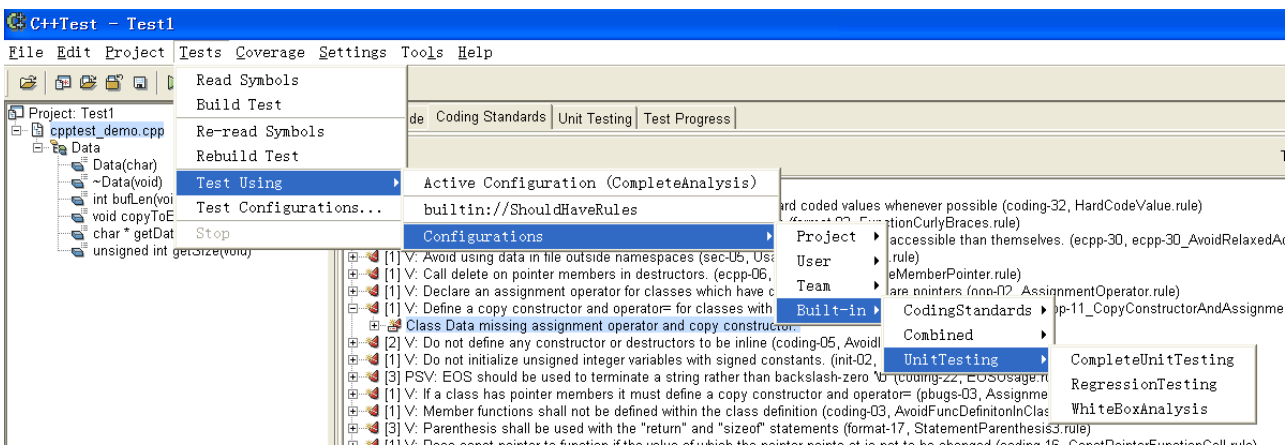
在静态分析栏中的 **Results** 标签是对静态分析结果的一个罗列。每个红色精灵帽都代表一种违规行为，而它旁边的数字则代表测试代码中出现这种违规的次数。紧接着的字母表明违规行为的严重级别。再后面就是对这条规范的大致描述 以及规则编号。

而标签 **Rules Manager** 则是对这些规则的管理，当用户需要使用某条规则的时候，只需要在相应规则左侧的方框内打上勾就表明选择了该条规则。而当用户不需要某条规则检查的时候，只需要去掉相应规则的勾就可以了。



### 3.4 动态测试

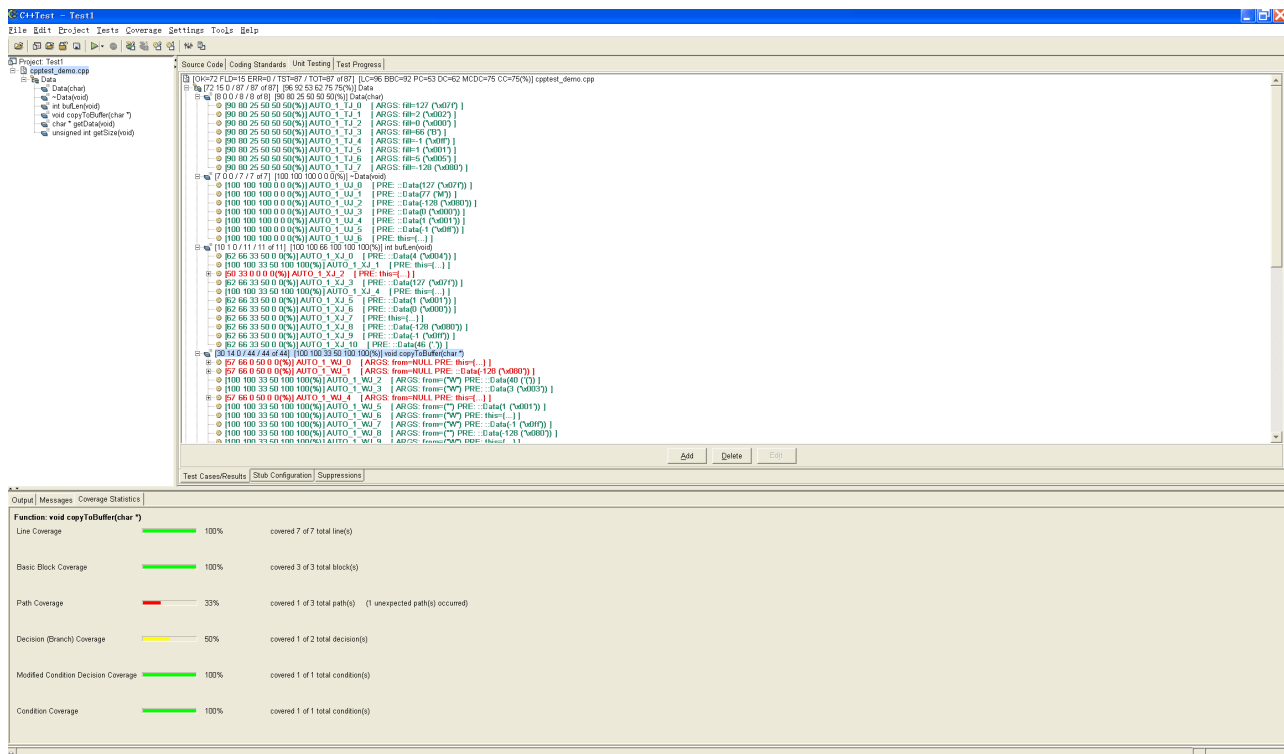
在向右三角型旁边的下拉箭头，选择内置的白盒测试。



C++ Test 将自动完成代码的动态测试。在动态测试中的 **Test Case/Results** 栏中，主要是对测试用例的一个总体管理。而 **Stub Tables** 栏则是对桩函数的管理，**Suppressions** 则是对测试对象的一个管理。

例如，上面的 **Data** 类有很多个成员函数，当用户并不想全部都测，而只是测其中的几个。这个时候就可以通过 **Suppressions** 进行选择。

**Tree filter** 还提供强大的过滤器功能，可以让用户更好的关注他们的焦点，例如只看最近一次测试的失败用例。

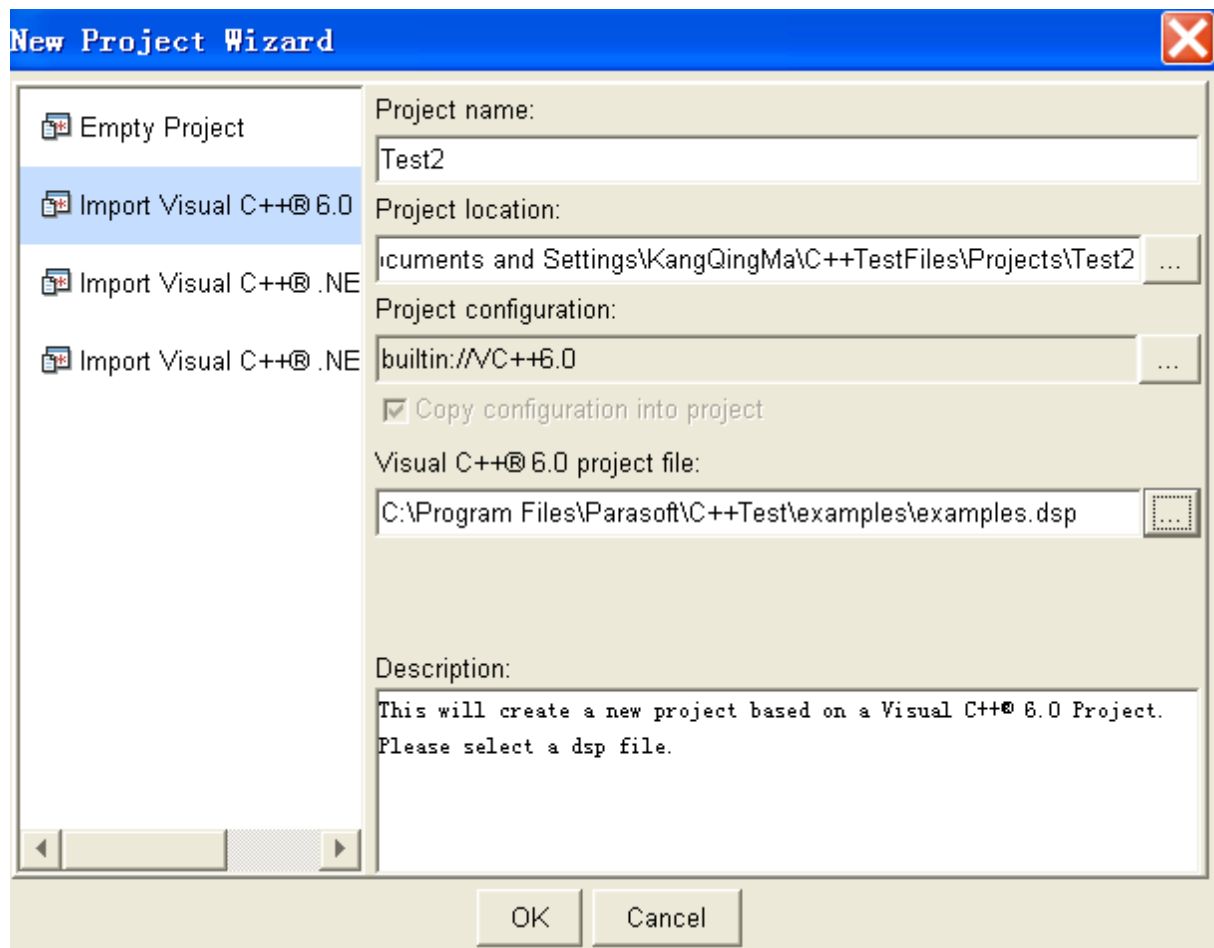


### 3.5 生成报表

选择 `file/generate report`，根据对报表的需求，选择适当的报表内容种类（例如dynamic analysis1,coverage statistic1）。

### 3.6 C++ Test 高级功能

C++ Test 6.0 可以直接导入 VC++ 6.0 project, VC++ .Net project 和 VC++ .Net2003 project 文件。点击 **File -> New Project**, 选择 **Import VC++ 6.0 project**, 输入测试工程文件名称, 选择 VC++ 6.0 工程文件.dsp,



如果你同时安装了 VC++6.0 和 VC++.Net 的话，你必须选择你需要的编译器。否则 C++Test6.0 会使用默认的编译器。

C++Test 6.0 可以选择不同的测试配置选项对一个工程、文件进行测试。点击 [Tests -> Test Configurations...](#) 打开测试配置界面，选择 [project -> 鼠标右键 -> New Configuration](#)。

[Analysis Flow](#) 决定是否要做编码规则测试和单元测试。

[Analysis Settings](#) 包括 [Coding Standards](#) 和 [Unit Testing](#)。其中 Coding Standards 选择需要的规则。打勾选择规则。

[Unit Testing](#) 中 [Types](#) 设置不同数据类型在生成测试用例的取值，可以增加某种数据类型在生成测试用例时的取值。

### 3.7 测试用例分析

用C++Test 做单元测试，最重要的步骤是分析测试用例。选择 [Examples\complex.cpp](#) 文件。





```

}
Complex testComplexOperators(Complex &a, Complex &b)
{
    Complex zero;
    Complex neg_a = zero - a;
    Complex neg_a_sum_b = neg_a - b;
    return a + b + neg_a_sum_b;
    // this function should always return complex zero value
}

```

我们的测试用例是针对函数 `Complex Complex::operator+(const Complex& c)` 的。从这里可以看出，上面测试用例测试后的结果是失败的。预期的结果是 `(_re=1,_im=0)`。实际的结果是 `(_re=3,_im=-3)`。

### 3.8 调试测试用例

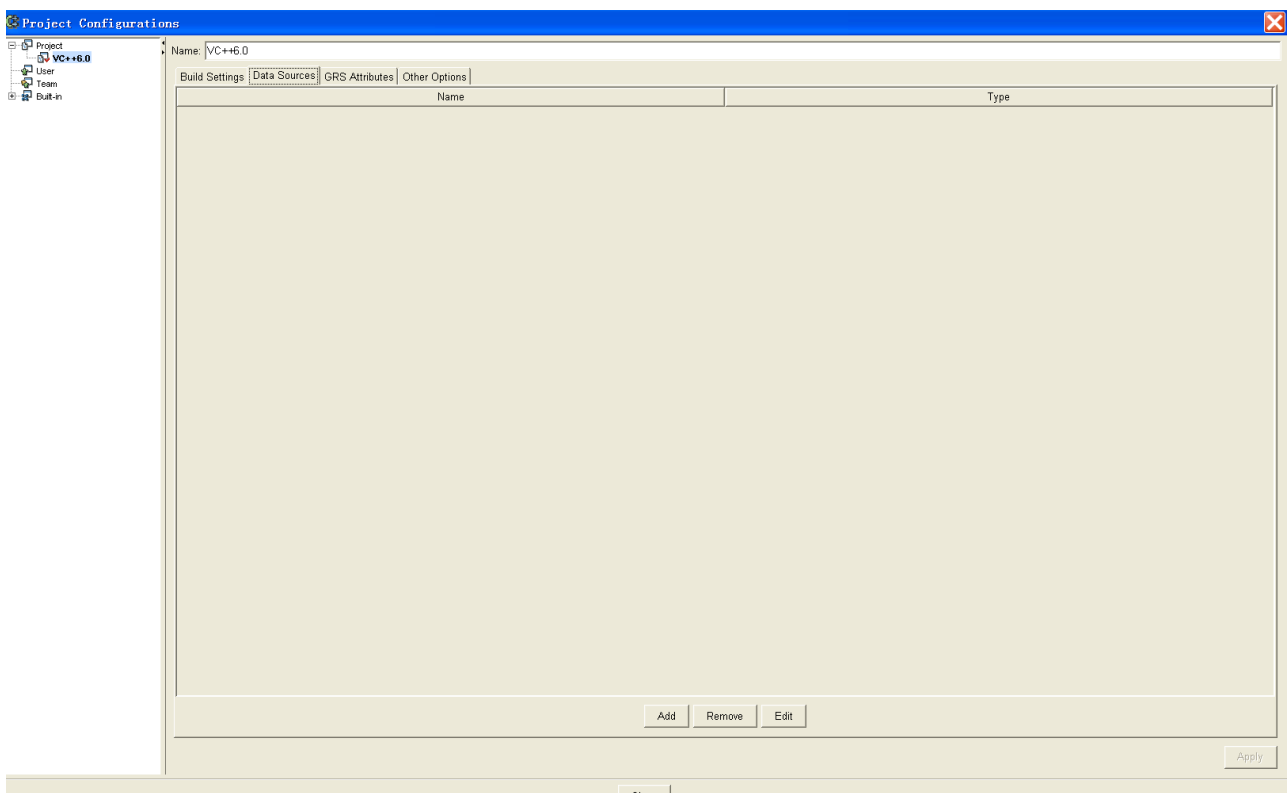
选择某个测试用例 -> 鼠标右键 -> Debug Selected Test Case(s) -> 打开VC++.Net 2003(默认编译器)调试界面。

注意：对VC++6.0好象无法进行调试。

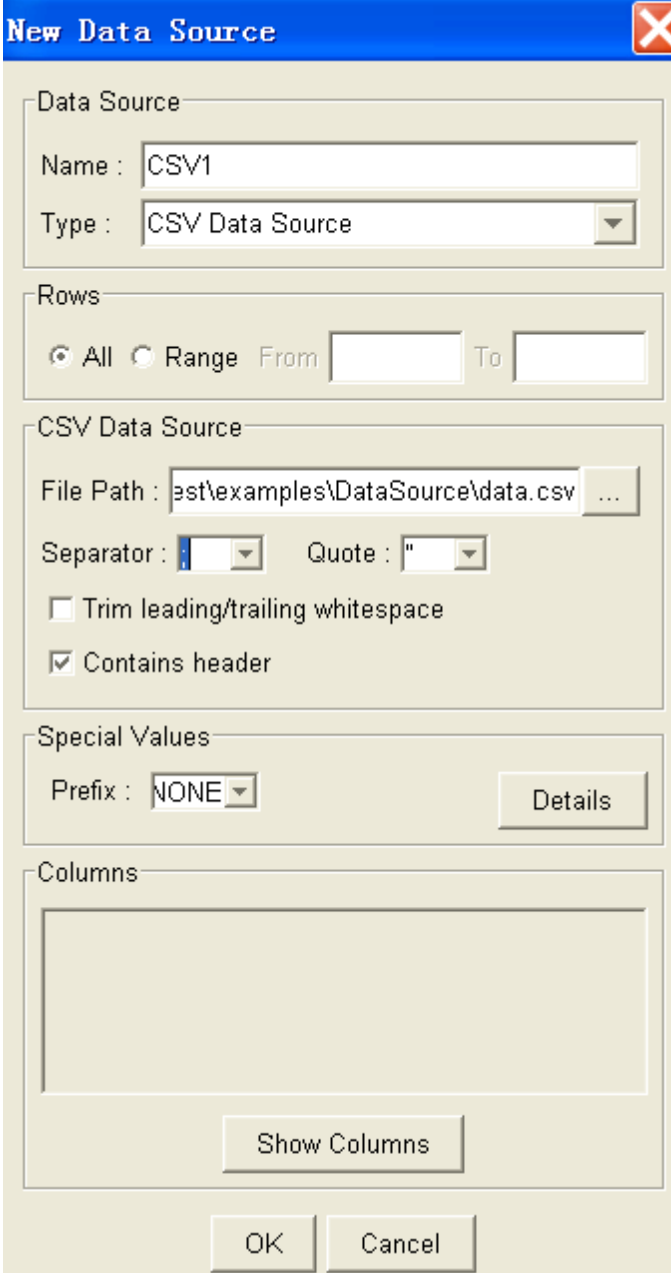
### 3.9 Data Source

如果一个函数有一系列相似的输入，比如协议类函数。这个时候可以把这一系列相似的输入保存成一个 excel 文件，CSV文件或数据库的表。我们称之为Data Source。当 C++Test 进行单元测试的时候，可以根据 Data Source 的内容生成一系列的测试用例。

打开 C++Test 安装目录 `Examples\DataSource\dsexample.cpp`，接着点击 `project -> project configurations...` -> 打开工程配置界面。



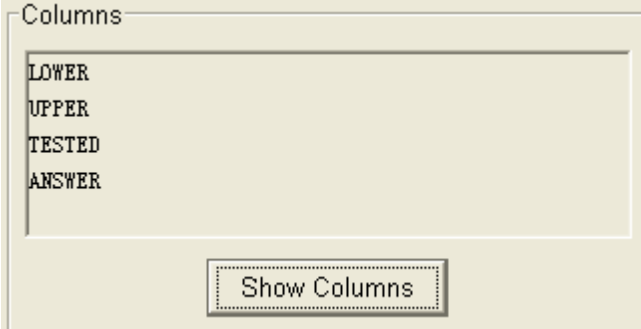
选择 **Data Sources**。按“Add”按钮增加 Data Source 文件。输入 Data Source 名称，选择 Data Source 类型为 CSV Data Source，选择 C++ Test 安装目录 **Examples\DataSource\data.csv** 文件。选择 data.csv 文件的分割符号。要根据实际的 .csv 文件分割符号选择 **'** 或 **;**。



The "New Data Source" dialog box is shown with the following settings:

- Data Source**
  - Name: CSV1
  - Type: CSV Data Source
- Rows**
  - ☒ All ☐ Range From: [ ] To: [ ]
- CSV Data Source**
  - File Path: test\examples\DataSource\data.csv
  - Separator: ; Quote: "
  - ☐ Trim leading/trailing whitespace
  - ☒ Contains header
- Special Values**
  - Prefix: NONE
  - Details button
- Columns**
  - Show Columns button
- OK and Cancel buttons at the bottom.

按按钮 **Show Columns** 显示 Data Source 文件的列名称。类似excel表格中的列头名称。

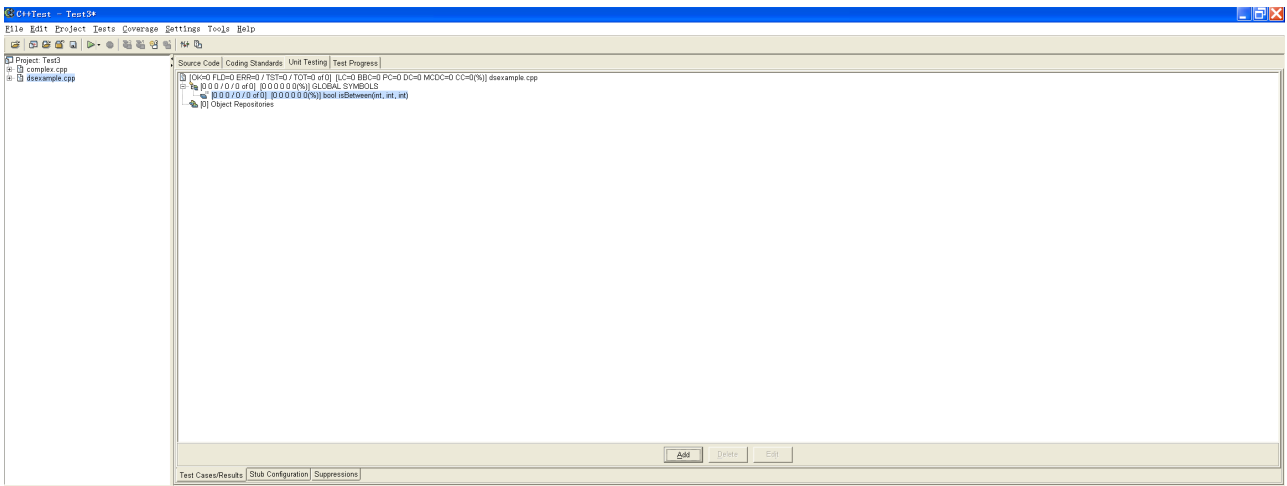


The "Columns" list shows the following column names:

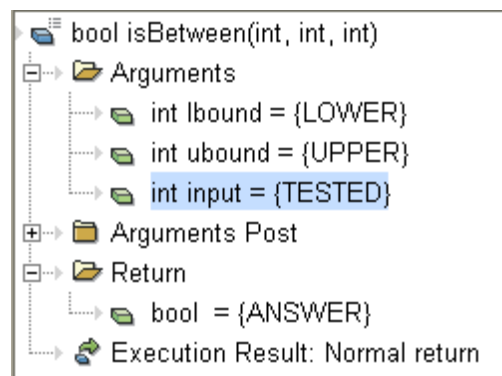
- LOWER
- UPPER
- TESTED
- ANSWER

Below the list is a **Show Columns** button.

根据设置的 Data Source 生成测试用例。选择 **Unit Testing -> 选择函数 isBetween -> Add**，增加测试用例。



选择刚才设置好的 Data Source 配置 `csv1`。要在 `Use Data Source` 前面打√。然后按 `Show Columns`。在输入、输出参数和返回值中选择对应的列。



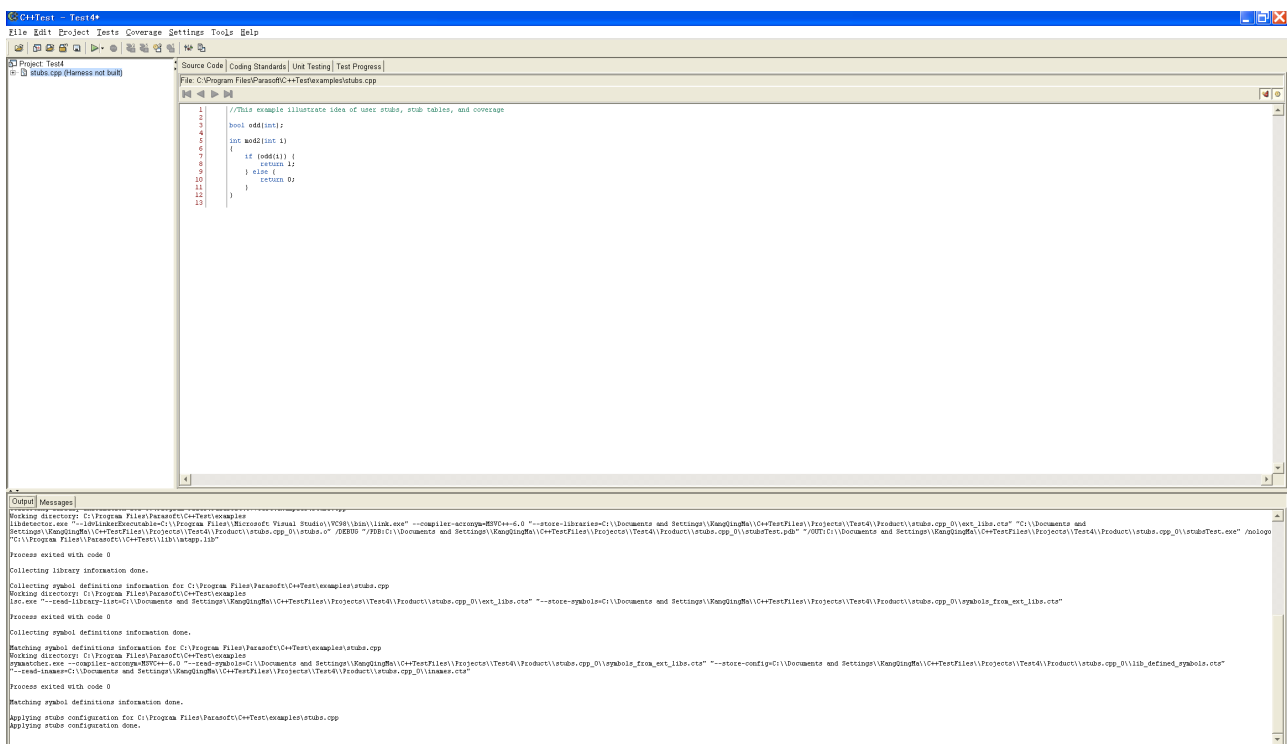
此时，测试用例将按照 Data Source 文件自动生成。

### 3.10 桩函数设置

对于单元测试，如果一个函数A调用了其他函数B（桩函数），而函数B由于还没有实现或其他原因无法使用。要正确测试函数A，就必须对函数B（桩函数）进行设置特定的返回值供测试函数A使用。

打开 C++ Test自带的 `Examples\stubs.cpp`。选择 `stubs.cpp -> 鼠标右键 -> Read Symbols` 解析 `stubs.cpp`。

从下面的界面可以看到 `mod2()` 函数调用了 `odd()` 函数。但是 `odd()` 函数没有实现。要对测试 `mod2` 的所以必须实现桩函数 `odd()`。



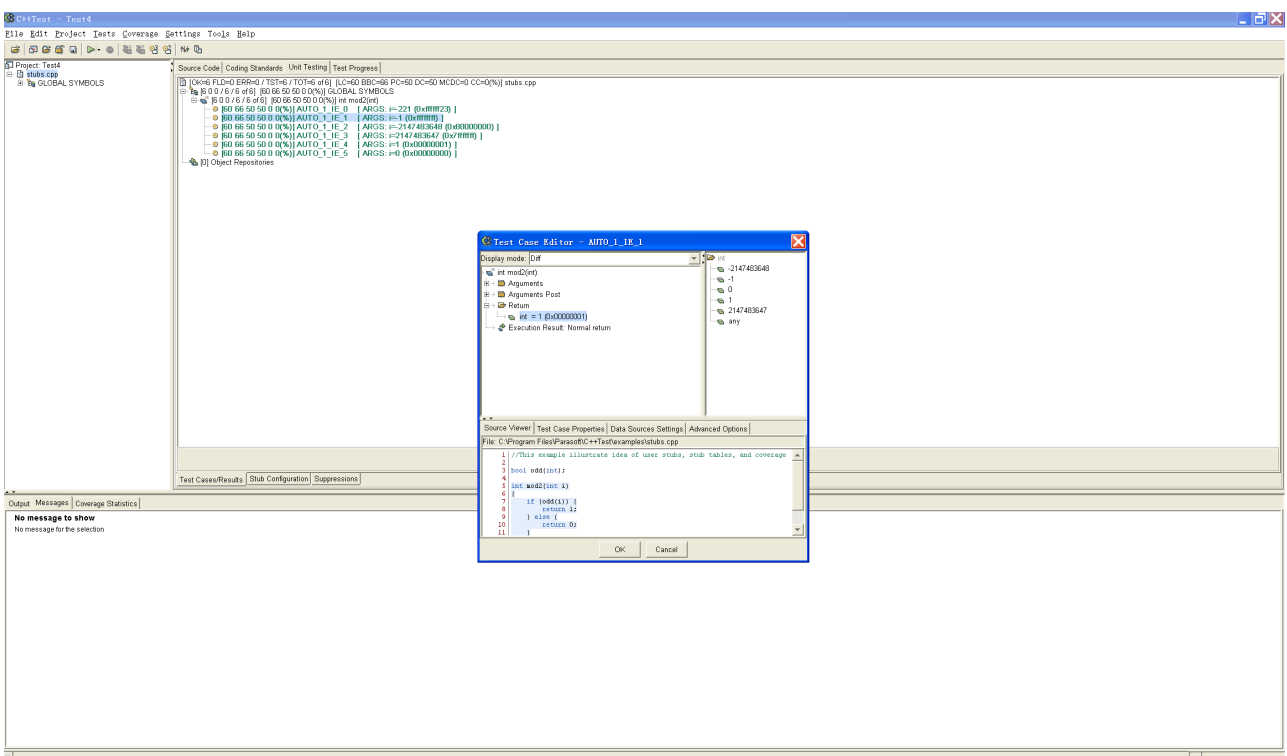
选择 Unit Testing -> 选择 Stub Configuration -> 选择 odd()函数 -> 鼠标右键 -> add user definition。



打开 stub 设置界面。人为增加代码 `return ture;`，让函数 `bool odd(int)` 返回 `true or false`（下面返回值为`true`），保存修改后的桩函数。



进行单元测试。打开 C++Test 测试用例编辑界面。不论输入 `mod2()` 函数的输入参数是什么值，它的返回值应当是1，这是因为桩函数 `odd()` 返回值为 `true`。



### 3.11 导入导出测试用例

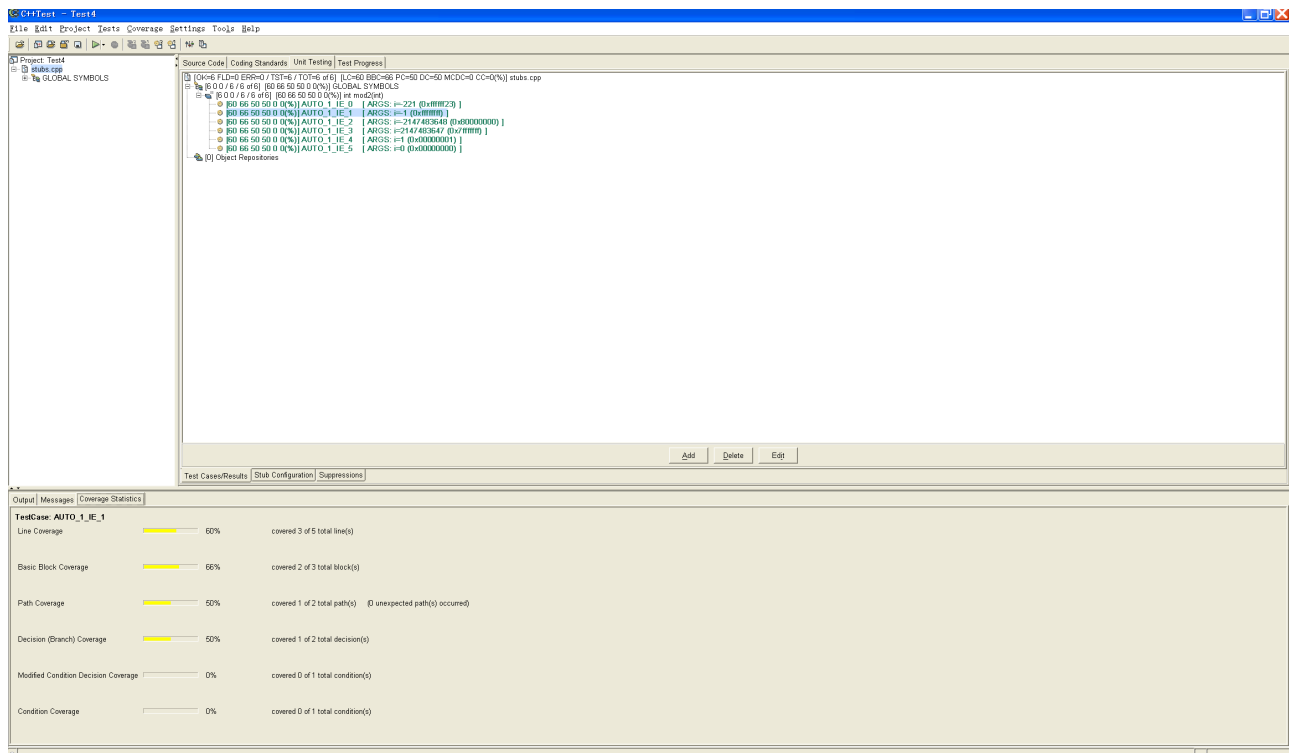
当你测试某个单元，使用一些特殊或自己定义的测试用例。而其他的人也同样测试此单元，你可以通过测试用例的 Import 和 Export 功能，导入或导出特定的测试用例。生成一个 XML 文件。

## 3.12 Test Objects

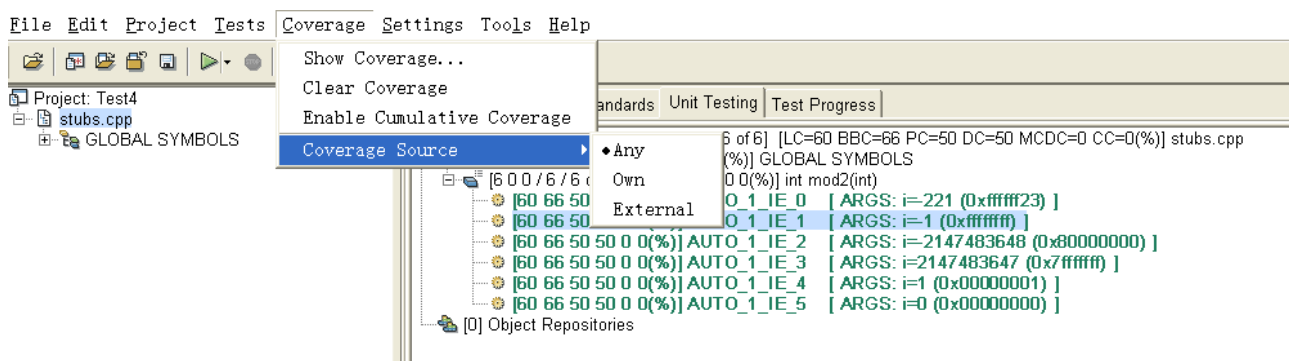
当你测试某个类的一个单元时，使用一些特殊或自己定义的类型构造对象。而测试一个类中的其他函数或其他人也需要此构造对象的时候，你可以通过 [Test Objects](#) 例的 [Import](#) 和 [Export](#) 功能，导入或导出特定的Test Objects。生成一个XML文件。

## 3.13 覆盖率分析

当分别选择工程、类、函数和单个测试用例的时候，在下边的Coverage Statistic Tab 中显示相应的覆盖率。



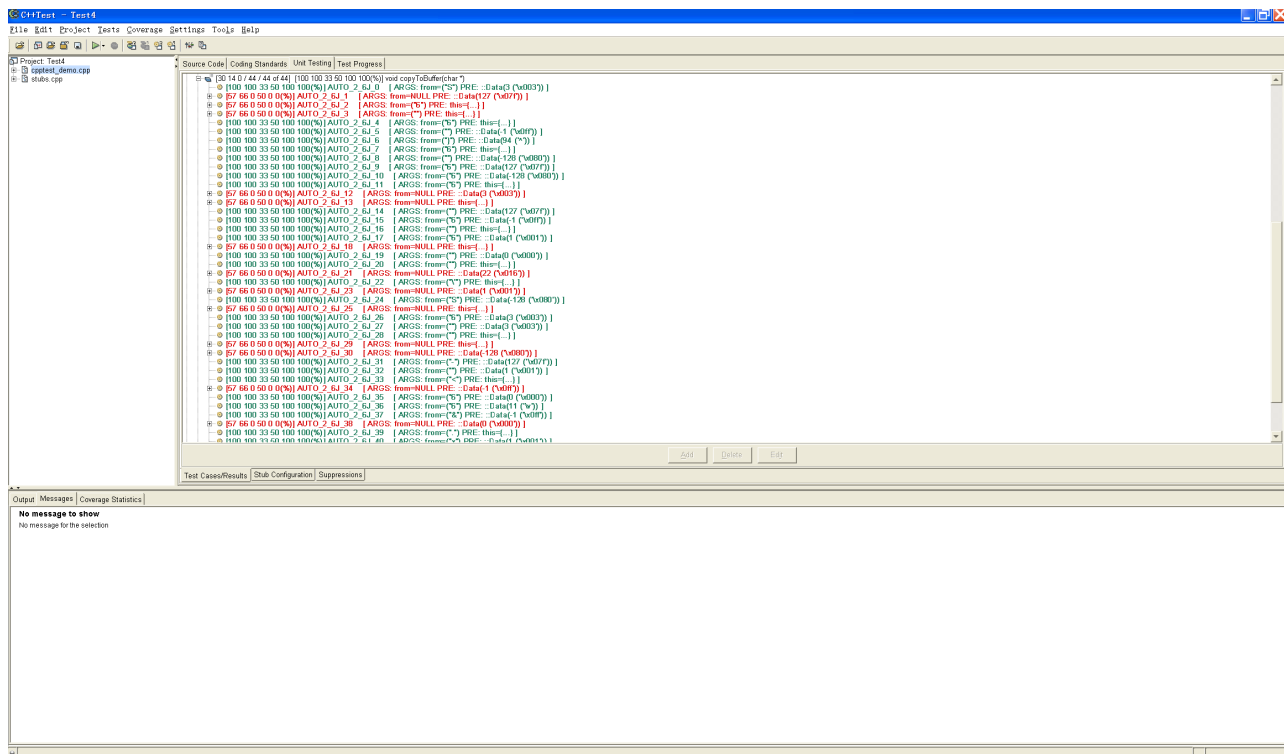
此外，从菜单 [Coverage -> Coverage Source](#) 可以选择覆盖率来源。



在左边的树状中选择文件，类或函数。然后选择菜单 [Coverage -> Show Coverage...](#) 打开源文件覆盖率。

## 3.14 回归测试

打开 [Example\cpptest\\_demo.cpp](#)。对其进行单元测试，结果如下：



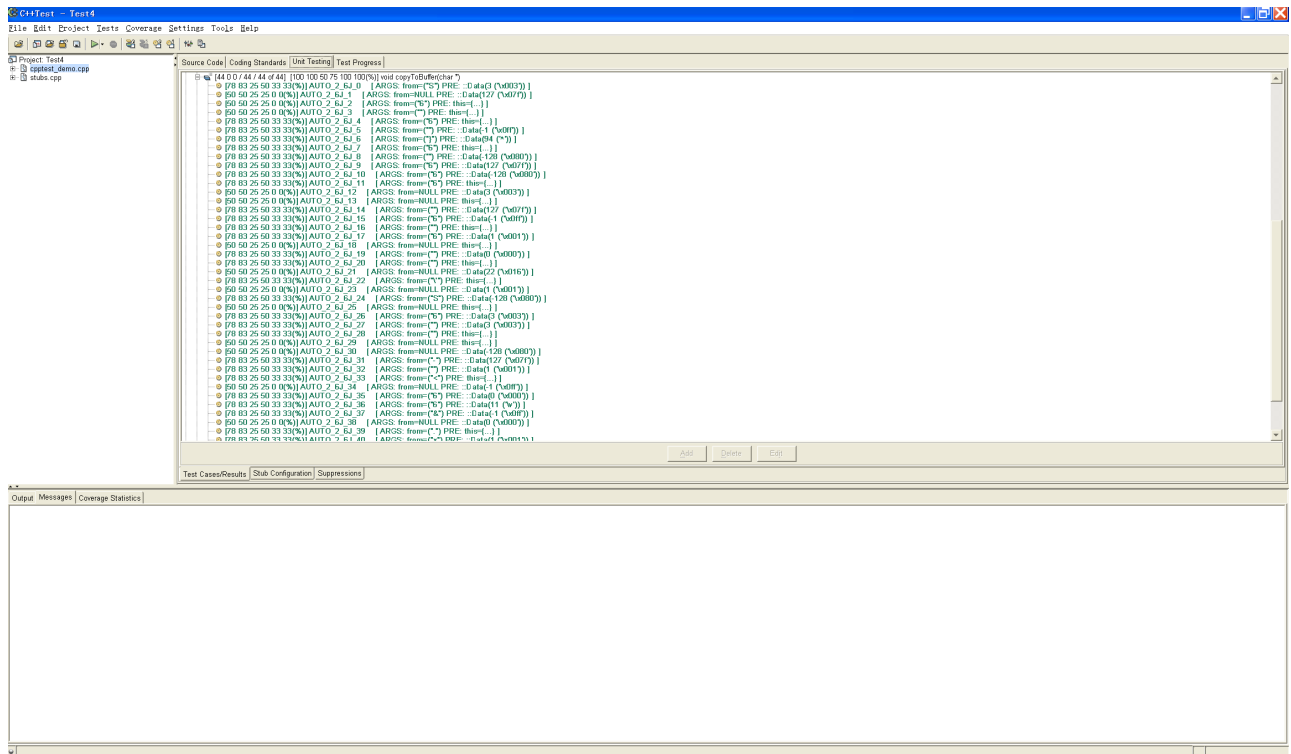
修改源代码

```
void Data::copyToBuffer(char *from)
{
    // argument should be validated - exception thrown if NULL passed
    // off by one error - should use '<' instead of '<='
    const unsigned SZ = getSize();
    for (int i = 0; i <= SZ; ++i)
    {
        *(_data + i) = *(from + i);
    }
    _data[SZ - 1] = '\\0';
}
```

为

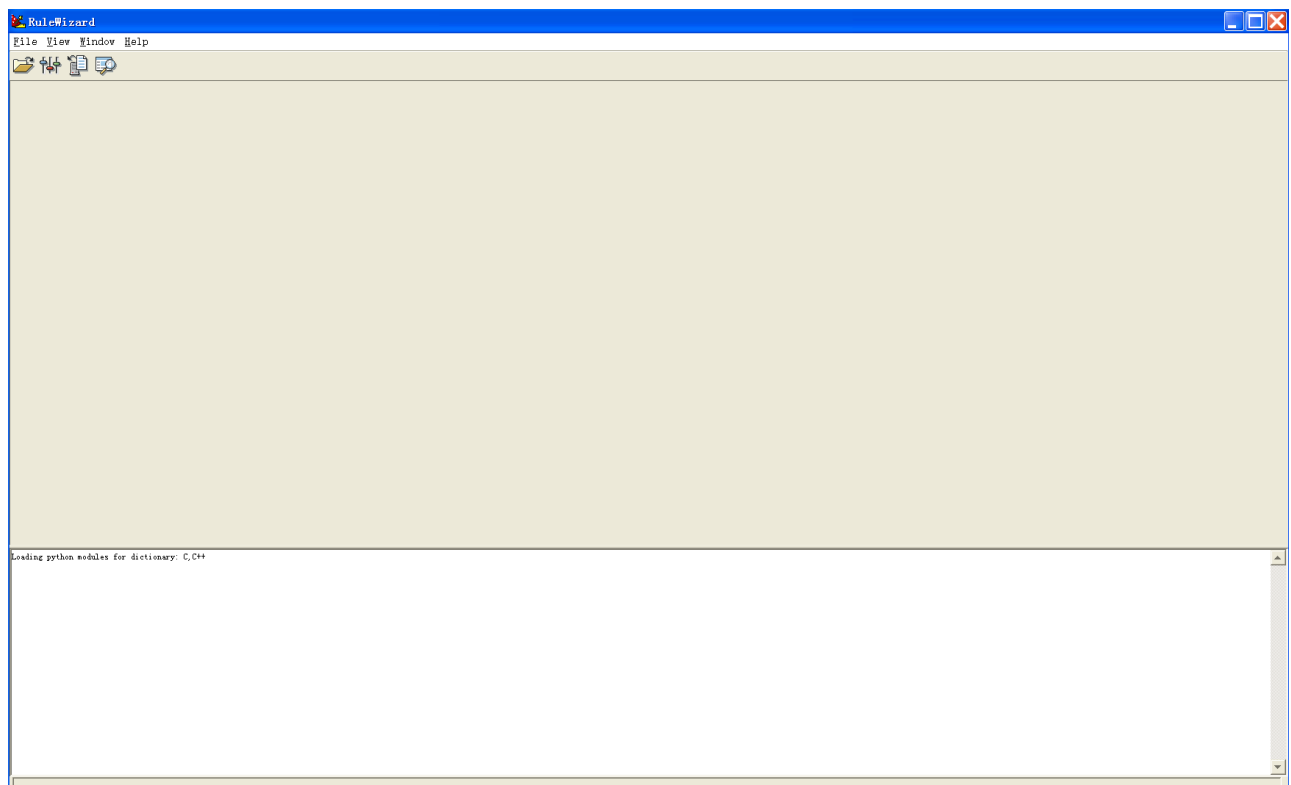
```
void Data::copyToBuffer(char *from)
{
    // argument should be validated - exception thrown if NULL passed
    // off by one error - should use '<' instead of '<='
    const unsigned SZ = getSize();
    if (NULL != _data && NULL != from)
    {
        for (int i = 0; i <= SZ; ++i)
        {
            *(_data + i) = *(from + i);
        }
        _data[SZ - 1] = '\\0';
    }
    else
    {
        _data = NULL;
    }
}
```

然后重新执行回归测试（不再重新生成测试用例，而是用原来的测试用例），结果如下：



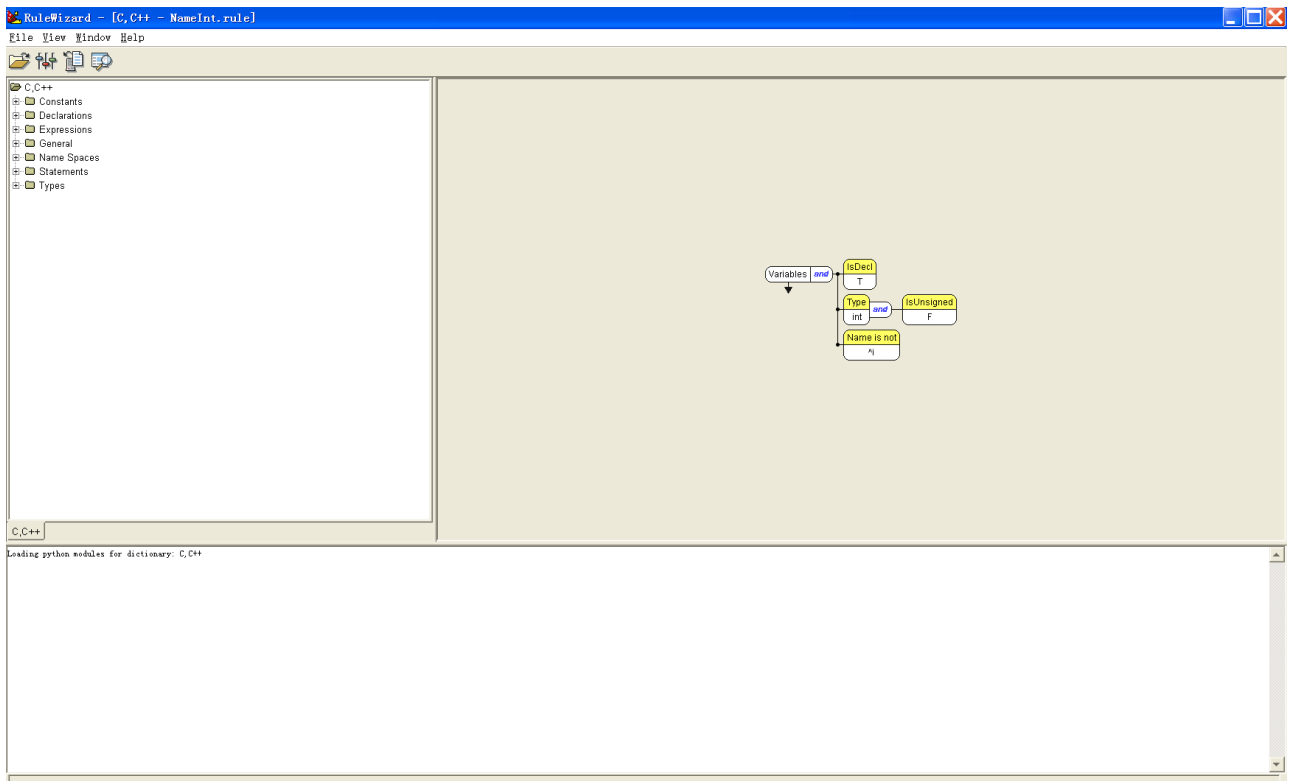
### 3.15 RuleWizard 定制规则

从 C++Test 主界面菜单 **Tools -> RuleWizard** (License 中必须包含有 RuleWizard 的功能 选项)。出现下面的界面 RuleWizard。



选择 RuleWizard 的菜单 **File -> Open** 打开 **C++Test\rules\naming\_conventions\NameInt.rule**。此规

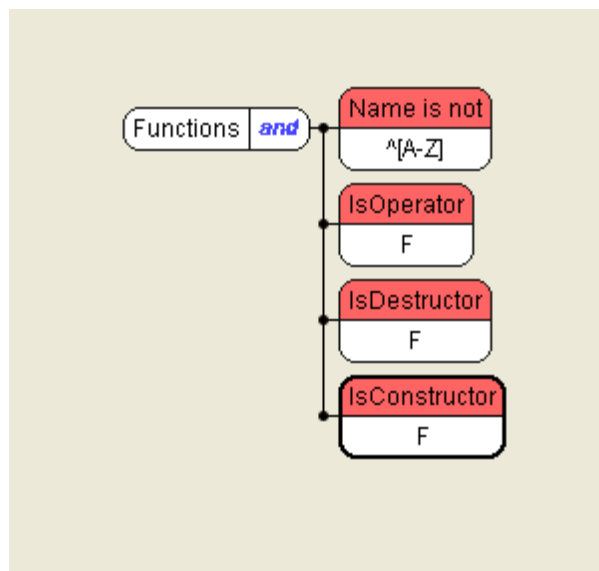




此外，还可以自己设计新的规则。菜单 **New...** 打开新规则界面，选择 **节点 C, C++ -> Declarations -> Functions**。然后按“OK”。

选中 **Functions 节点 -> 鼠标右键 -> Names(s)**，在 RegExp 中输入 **^[A-Z]**，并且选择 **Negate**。表示：“函数名称必须大写字母开头，如果不是则报错”。

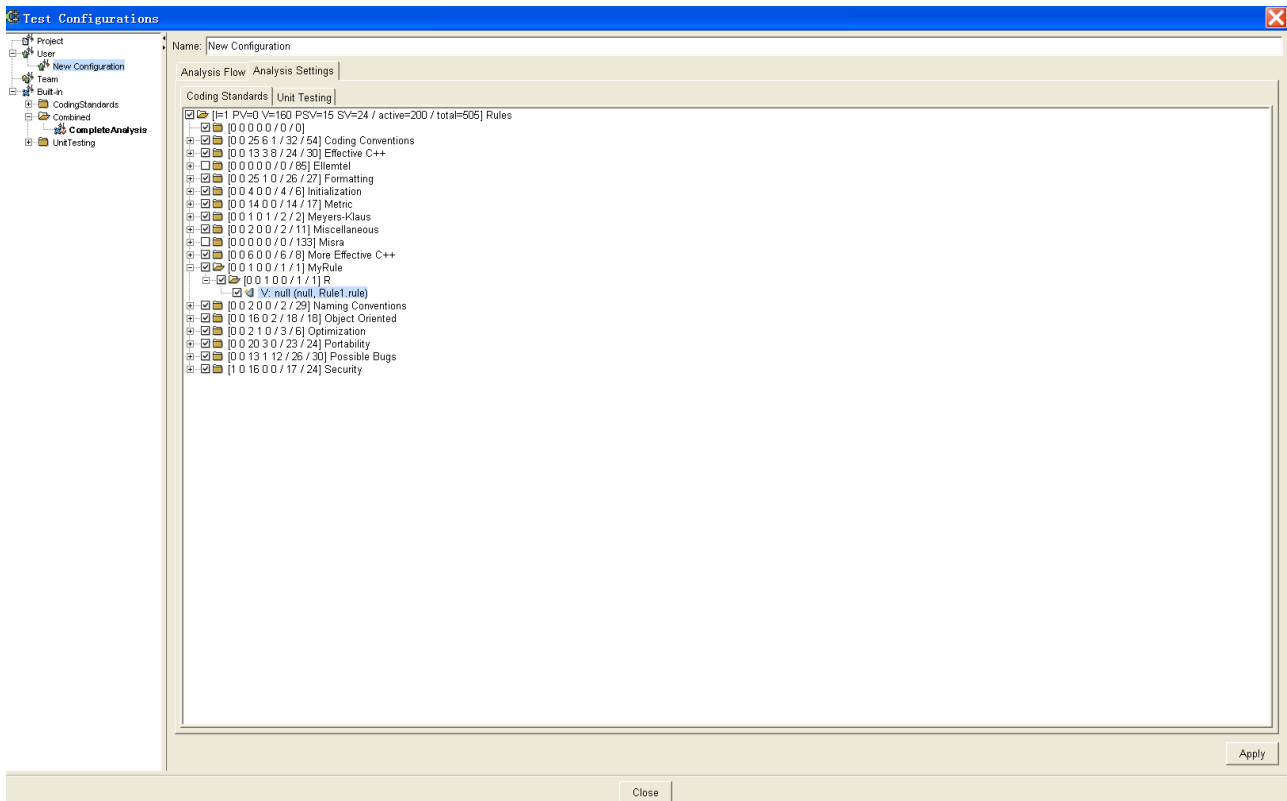
选择 **Functions 节点 -> 鼠标右键 -> 选择IsOperator(F/T)**。表示：“函数不是操作符号”。双击 **IsOperator** 的 **T** 开关，自动变成 **F**。用同样的方法分别增加 **IsConstructor** 和 **IsDestructor**。并且都设置为 **F**。表示：“函数不是构造函数，也不是析构函数”。



设置检查结果显示标题，选择 **节点Functions -> 鼠标右键 -> Create Output -> Display**。在 **Message** 中输入“A function name should begin with a capital letter” 函数必须大写字母开头。

在 **空白处 -> 鼠标右键 -> 选择Properties**，可以检查规则。保存规则到自己的目录，给规则取个好理解的名字 **FunctionsNameCapital.rule**。

接下来，打开测试配置界面。在一个 **Test Configuration**（比如 User -> New Configuration），选择 **Analysis Settings** Tab -> **Coding Standards** Tab -> 选择根目录 -> 鼠标右键 - Add Rule Set...。选择自定义规则。



至此，C++ Test 的基本功能就结束了。