BMOCK

User Guide

By Asher Sterkin
Principal Engineer, NDS Technologies Israel
e-mail: asterkin@nds.com, asher.sterkin@gmail.com
Last update : 19/12/2007

1 Overview

The **bmock** (tentatively stands for Boost Mock) is a C++ library supporting Mock objects (www.mockobjects.com). The **bmock** library is tightly integrated with the Boost.Test unit testing library (www.boost.org). Conceptually the **bmock** library follows more or less the same approach as JMock (www.jmock.org) , NMock (www.nmock.org) libraries, but in its current form it follows a strict type checking approach suggested by the EasyMock (www.easymock.org) library. In many aspects the **bmock** library is quite different from another C++ mock library: Mockpp (mockpp.sourceforge.net) primarily because **bmock** by no means enforces a developer to work within a pure Object-Oriented paradigm. Originally the idea of the **bmock** library was triggered by a need for PC-based unit testing of consumer electronics software written in a plain "C" with a very arcane system APIs. For that reason **bmock** was from the very beginning developed using the link time resolution approach: you want to use a mock version of some function, all what you need is to supply this mock version at link time. For that reason mixing mock and real versions of the same function within the same project is not possible.

Since C++ does not support reflection generating mocks on the fly in run-time does not seem to be possible. Still, minimizing efforts spent on mocks creation is a must, otherwise nobody will ever use them. For that reason in the **bmock** library mocks are automatically generated from an IDL like function annotations like this:

```
BMOCK_FUNCTION(int,f,1,(IN(int,x)))
```

Basic support for IN, OUT, and IN_OUT arguments as well as a special treatment for raw memory (usually `void *`) buffers are provided. The **bmock** library supports generating mocks for plain "C" functions, and C++ member functions including constructors and destructors. C++ exceptions and general callbacks are supported as well.

The current version of the **bmock** library is available for Visual C++ 8.0 only, and is tightly integrated with the Visual Studio .NET 2005 IDE.

In addition to the **bmock** library itself this distribution contains the Boost.Statechart library (http://boost-sandbox.sourceforge.net/libs/statechart/doc/index.html) and a Visual Studio .NET 2005 project template for Python (www.pyhton.org) modules created with the Boost.Python (http://www.boost.org/libs/python/doc/index.html) library

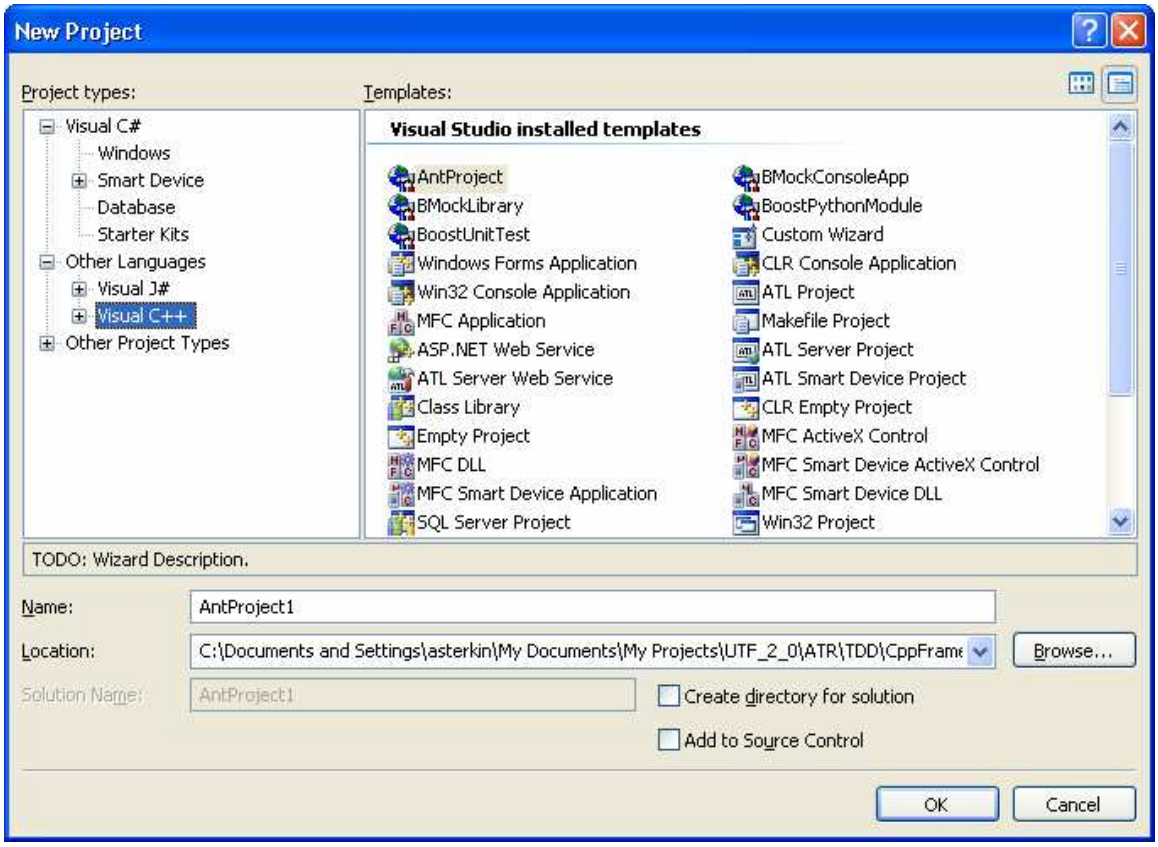
2 Installation

1. Unzip the **bmock_g_r_i_b.rar** file at C:\ drive (or where is you Visual Studio .NET 2005 is installed)

2. If you plan on using the Boost.Python template, add C:\Boost\lib to your PATH environment variable
3. Launch Visual Studio 2005

3 Working with *bmock* projects

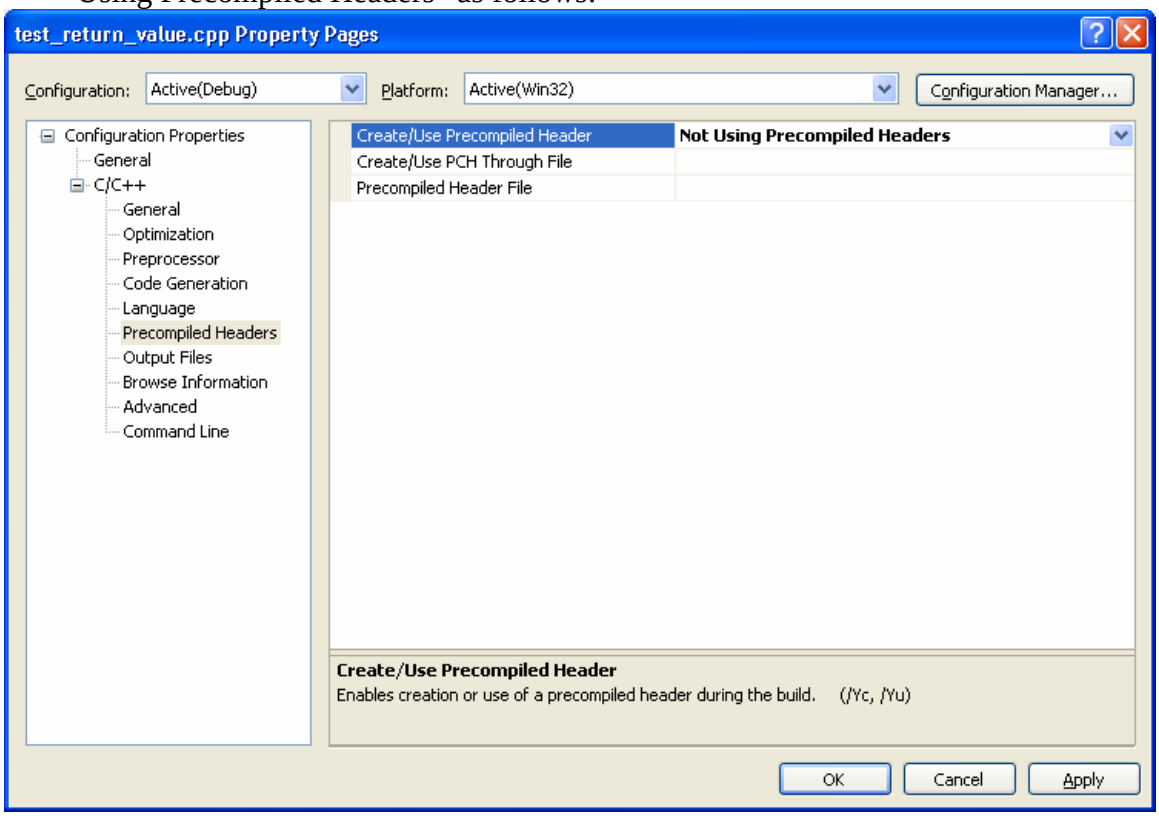
1. The **bmock** library comes with three basic project templates: BoostUnitTest for developing unit test applications, BMockLibrary for developing stand-alone mock libraries, and BoostPython for developing Python modules with the Boost.Python library.
2. Select “New Project” menu option.
3. In the “New Project” dialog box select Visual C++ and one of the following options: BoostUnitTest, BMockLibrary, BMockConsoleApp, BoostPythonModule, AntProject.



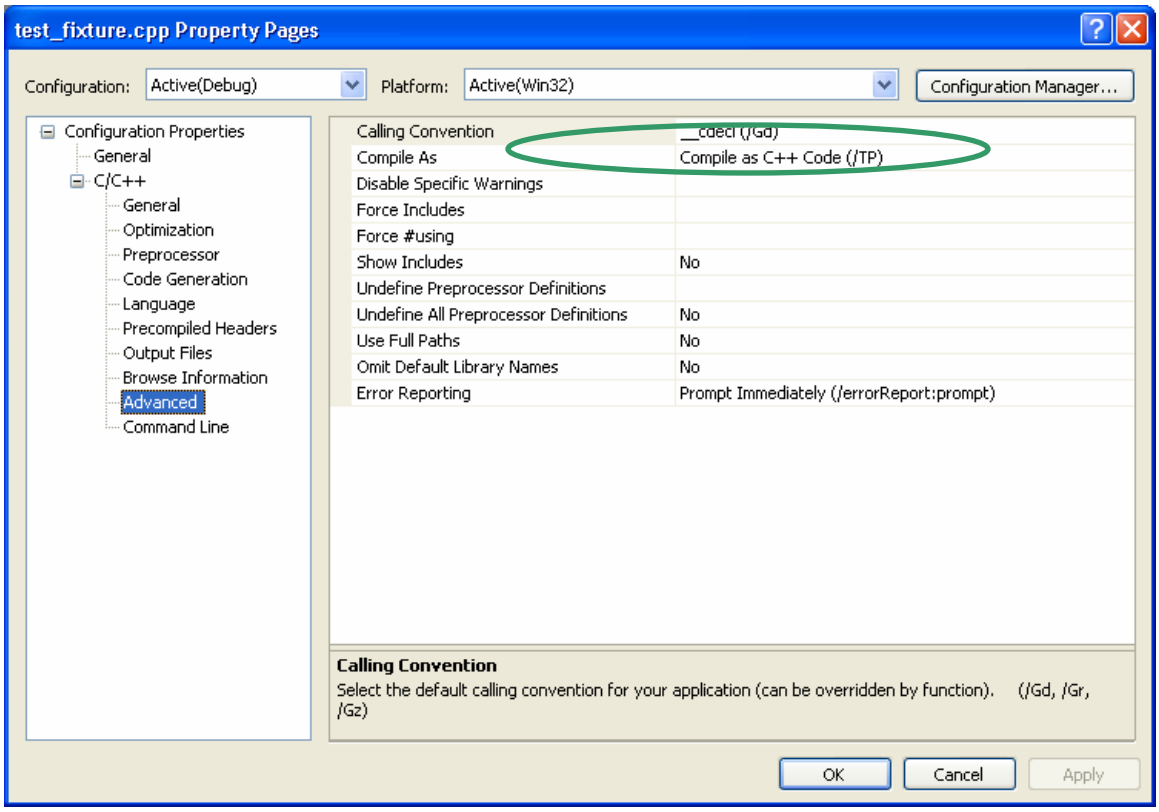
4. Normally you will work with either the BoostUnitTest or BMockLibrary type of projects.
5. The BMockConsoleApp project template supports building a console version of mocked application. This is an experimental feature intended to support integration of **bmock**-based units with Java version FitLibrary.
6. The BoostPython project template supports creating a Python module using the Boost.Python integration technology. The main reason for providing this template is

that it might be useful for integrating **bmock**-based units with the Python version of the FitLibrary.

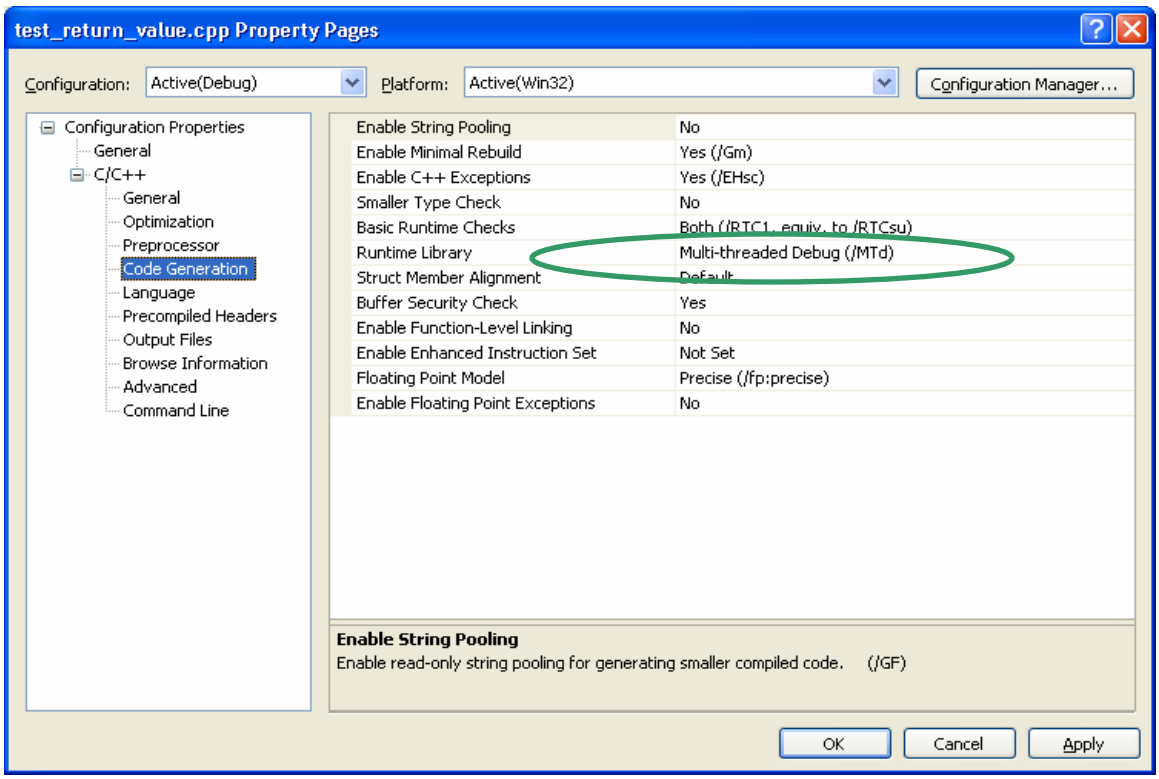
7. The AntProject project template creates an Apache Ant. This kind of project is mainly used for batch build and installation automation purposes.
8. Provide a name for you project and press OK.
9. A new C++ project will be automatically generated.
10. By default all **bmock** projects are configured for automatic creation and using pre-compiled headers. If you do not want to use precompiled headers for your target source code files, you will need to set manually the file's Properties/C++/Precompiled Headers/Create-Use Precompiled Header to "Not Using Precompiled Headers" as follows:



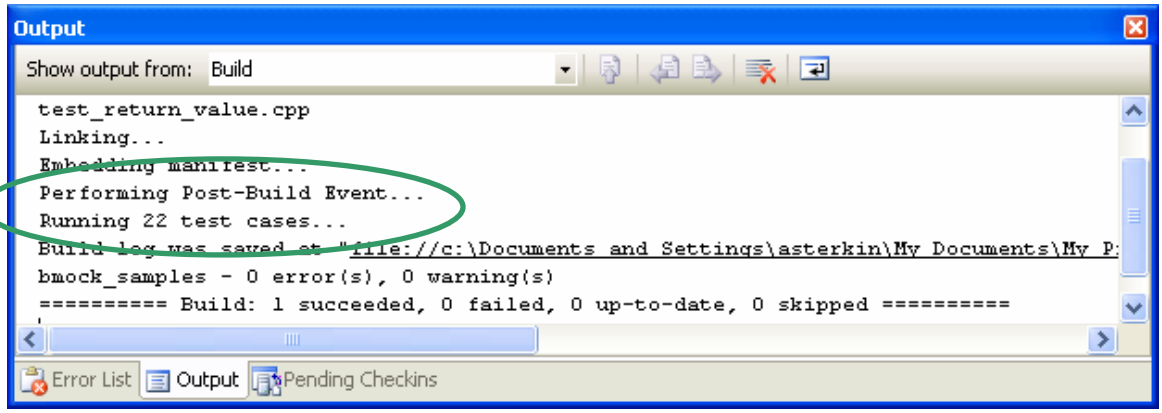
11. If you need to include "C" source files into your project it is recommended to set their "Compile As" to C++ as follows:



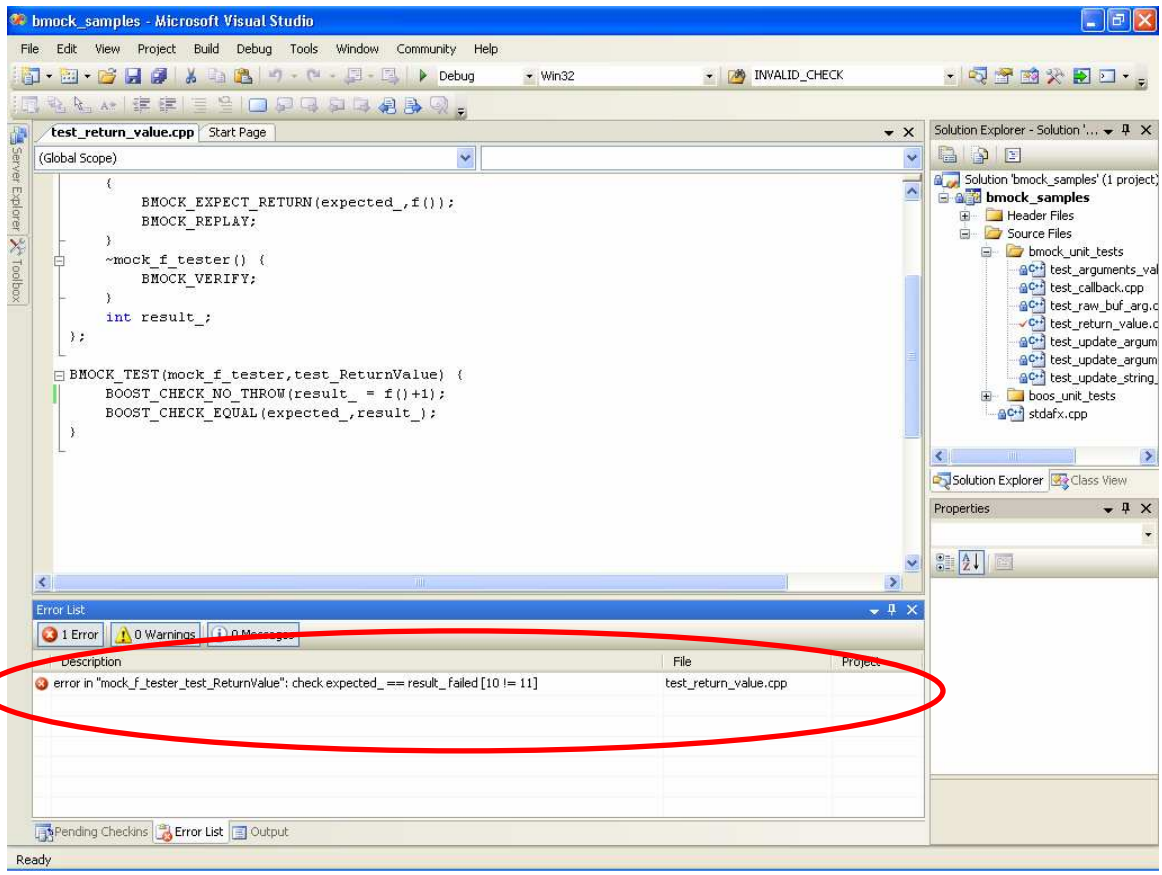
12. C++ code generation for the both BoostUnitTest and BMockLibrary projects is set to use a Multithreading Static version of the C++ run-time:



13. You SHALL NOT change this setting.
14. The BoostUnitTest projects are configured to run tests automatically as a Post-Build event. If all tests are passing you will see in Visual Studio Output window a summary message like this:

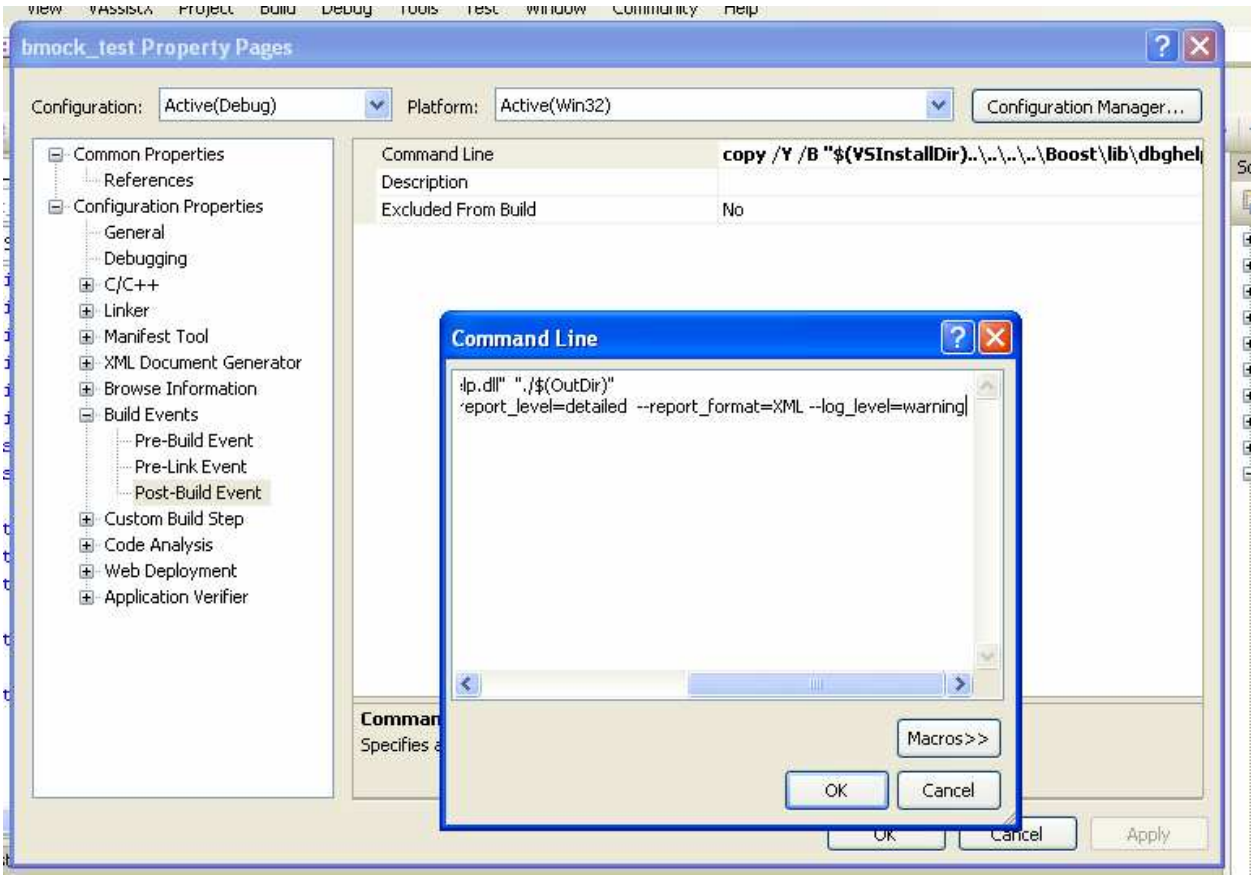


15. If one or more tests failed you will get an compiler-like error message in the Visual Studio Output and Error List window as follows:

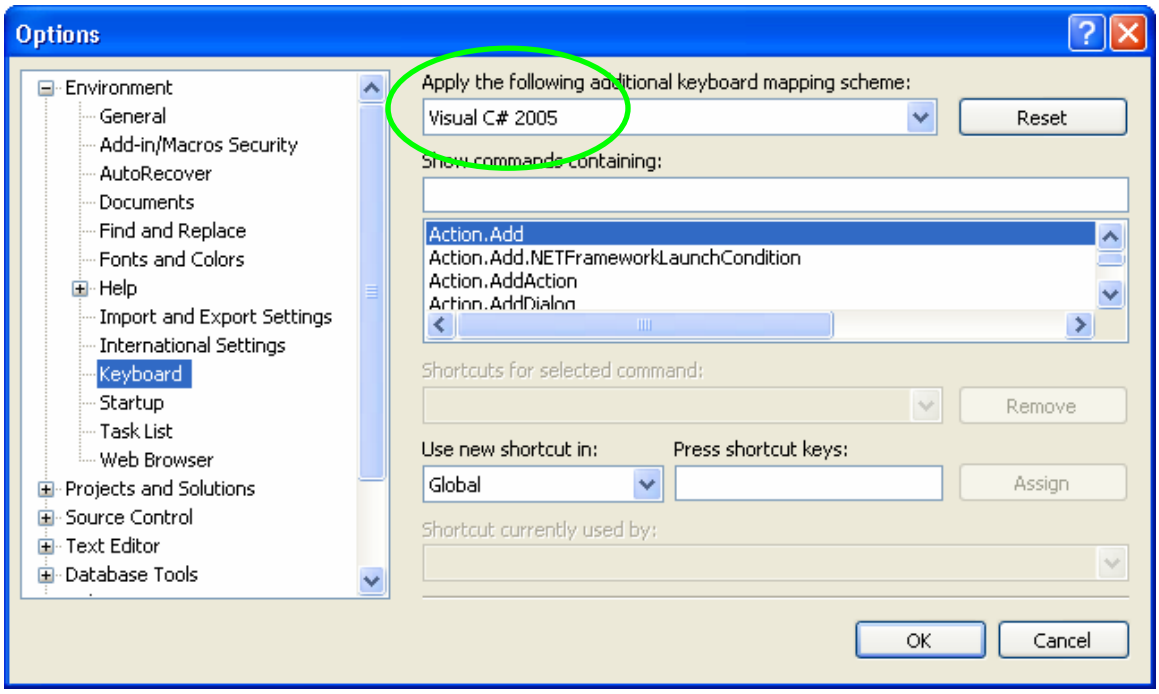


16. A double click on an error message line will bring you to the source code line where a particular test failed.

17. In earlier versions of bmock the BoostUnitTest projects does not support bmock and boost warnings, if you created the project with an old bmock version you can configure it to support the warnings by adding
 --log_level=warning
in the project's post build event command line, thus creating a command line that looks like this:
"$(OutDir)/$(TargetFileName)" --result_code=no --report_level=detailed --report_format=XML --log_level=warning



18. In order to compile and run your current test all that you need to do is to press CTRL-SHIFT-B if you want to build the whole solution or SHIFT-F6 if you want to build/run only the current test (might be handy for a large solution). To make SHIFT-F6 work properly ensure that your keyboard setting (Tools/Options/Keyboard) are set for Visual C# 2005 as follows:

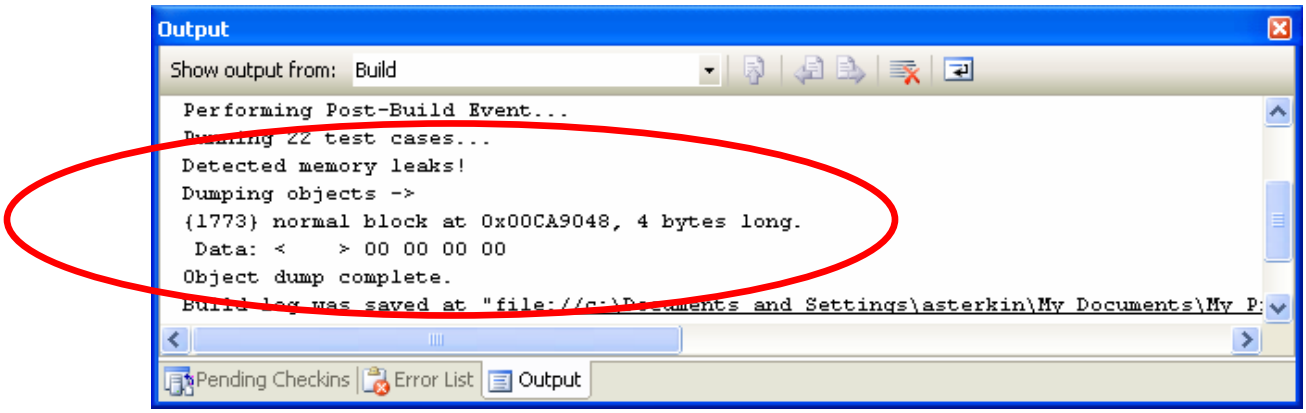


19. All project templates come with non-trivial stdafx.h and stdafx.cpp. Although you might add additional frequently used header files into a stdafx.h file, you SHOULD NOT modify or delete any of existing definitions – they all come on purpose.
20. The only exception from the rule stated above is when you need to include one or more header files coming with Microsoft Windows SDK (for example WinSock2.h). The right place to put these additional includes is within the stdafx.h file as follows:

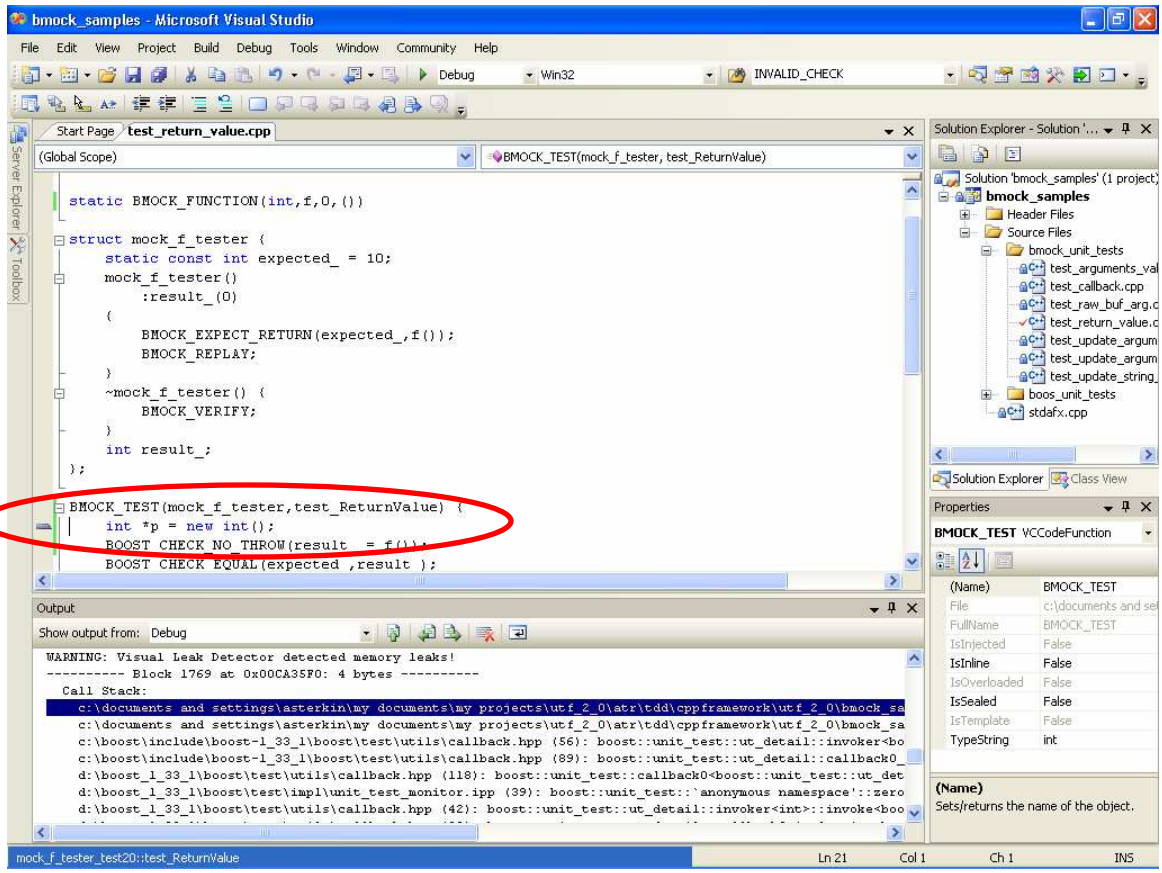
```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//
#pragma once
#define WIN32
#define _CONSOLE
#define WIN32_LEAN_AND_MEAN             // Exclude rarely-used stuff from
Windows headers
#define _CRT_SECURE_NO_DEPRECATE
#define _SCL_SECURE_NO_DEPRECATE
#include <vld.h>
#pragma warning (push)
#pragma warning (disable: 4267)
#include <boost/test/auto_unit_test.hpp>
#pragma warning (pop)
#include <windows.h>
#undef FAR
#define FAR
#undef PASCAL
#define PASCAL
//
// If required put other windows header files (e.g. WinSock2.h) here
```

```
//
#undef IN
#undef OUT
#include <bmock/bmock.hpp>
#include <bmock/bmock_util.hpp>
```

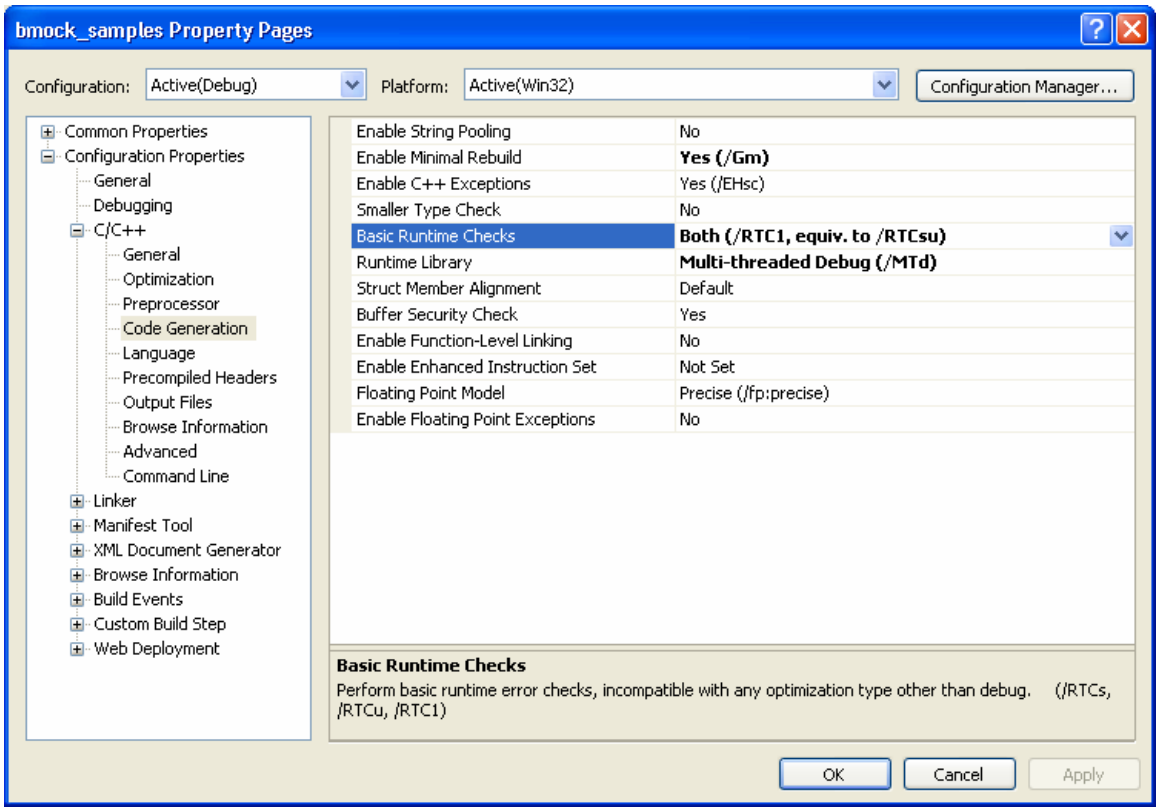
21. Normally you may keep your target source files as a part of the BoostUnitTest project, but sometimes you might want to isolate them into a separate static library. In order to avoid any project settings headache it's recommended to use the BMockLibrary template for this - there is nothing harmful. At any development it is recommended to build your final product using a separate build script or even a separate set of development tools (in consumer electronics it's usually just out of the question).
22. The BoostUnitTest project template includes a memory leaks detection library called VLD (http://www.codeproject.com/tools/visualleakdetector.asp). If your tests do not leave dynamic memory unreleased, nothing will happen. But if there is some memory leak, the VLD library will print an error message as follows:



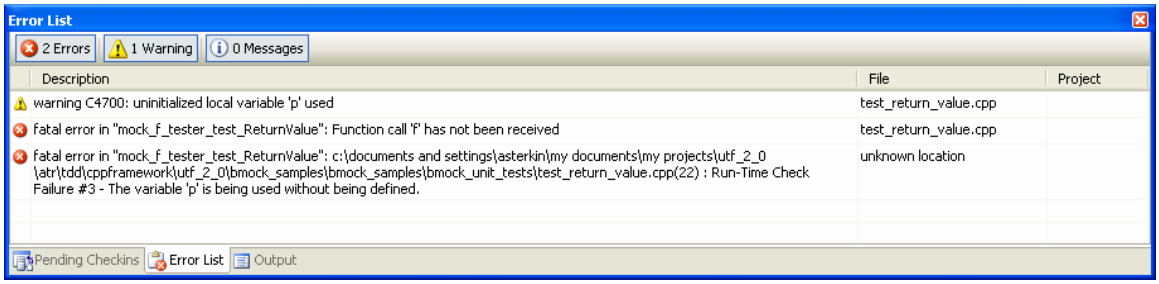
23. In order to get a more detailed diagnostics you need to run your unit test application with debugger (press F5). Then you will get the following:



24. Double click on the first reference line within the stack trace report will bring you to the corresponding source code location where unreleased piece of memory has been allocated.
25. The both BoostUnitTest BMockLibrary project templates are configured to perform all types of run-time checks in the Debug mode:



26. Unfortunately this kind of errors are not automatically printed when a test is being run as a Post-Build Event (see more about recommended policy for unit test exception handling below). Un-initialized variables are the most typical cause for run-time errors and the C++ compiler will normally issue a warning. The test itself will most likely fail with a fatal error like this:



27. For a more in-depth investigation run your unit test with a debugger.

4 Writing Unit Tests

4.1 Using the Boost.Test Library

1. The BoostUnitTest projects are configured for working with the BOOST_AUTO_UNIT_TEST facility. Except for this the rest of the Boost.Test library features are fully supported. For instance a simple test will look like:

```
#include <StdAfx.h>

static const int expected = 125;

static int f() {
	return expected;
}

BOOST_AUTO_UNIT_TEST(testF_Equal) {
	int result;

	BOOST_CHECK_NO_THROW(result = f());
	BOOST_CHECK_EQUAL(expected,result);
}
```

2. A more sophisticated test using the Boost lambda library and Boost.Test predicates will look like:

```
#include <StdAfx.h>
#include <boost/lambda/lambda.hpp>
using namespace boost::lambda;

static const int expected = 125;
static const int not_expected = 127;

static int f() {
	return expected;
}

BOOST_AUTO_UNIT_TEST(testF_NotEqual) {
	using namespace boost::lambda;
	int result;

	BOOST_CHECK_NO_THROW(result = f());
	BOOST_CHECK_PREDICATE((_1 != _2), (not_expected) (result));
}
```

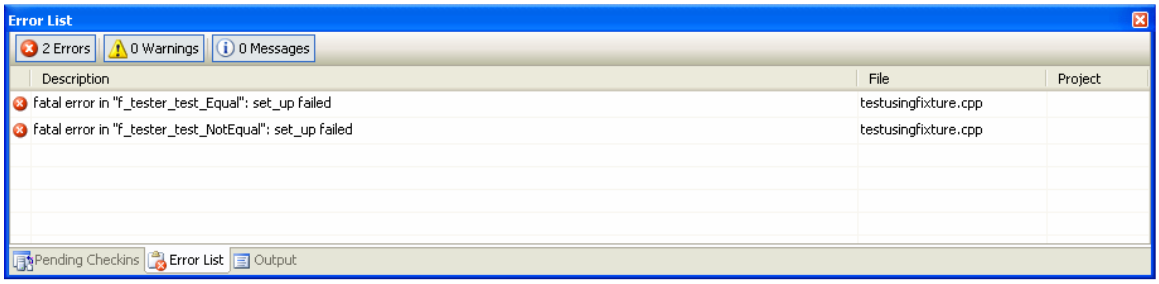
3. For complete documentation of the Boost.Test library consult http://www.boost.org/libs/test/doc/index.html.

4.2 Working with Fixtures

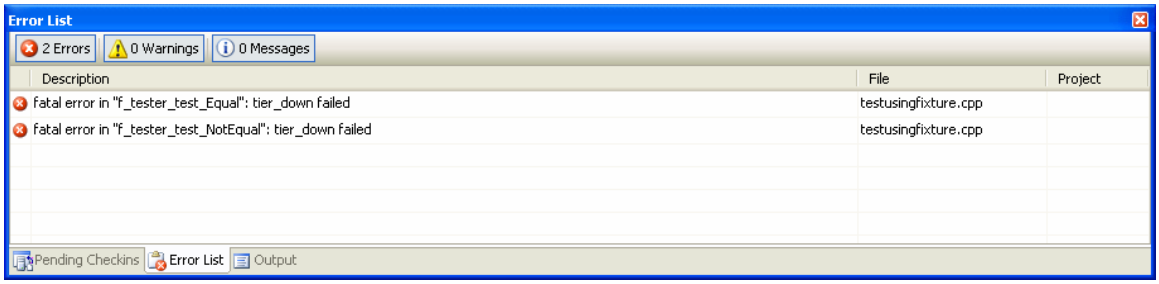
1. The **bmock** library comes with a built-in support for fixture classes similar to those of JUnit (www.junit.org) and NUnit (www.nunit.org) taking into consideration the lack of reflection support in C++. Simply saying, a fixture is a class, which constructor and destructor are automatically invoked at the beginning and at the end of EVERY test. This feature is typically required when a number of tests share the same initialization and cleanup procedure, but NOT the same state of member variables. The latter is considered as unhealthy practice by many unit testing gurus and is not recommended (although you can achieve this effect by using the Boost.Test library test case classes directly). A simple test using the **bmock** fixture will look like:

```
#include <StdAfx.h>

static const int expected_1 = 125;
static const int expected_2 = 127;

static int f(int &x) {
      x = expected_2;
      return expected_1;
}

struct f_tester {
      f_tester() {
            result_ = f(x_);
      }
      int x_;
      int result_;
};

BMOCK_TEST(f_tester,test_Equal) {
      BOOST_CHECK_EQUAL(expected_1,result_);
}

BMOCK_TEST(f_tester,test_NotEqual) {
      BOOST_CHECK_EQUAL(expected_2,x_);
}
```

2. This practice is also useful when we want to test multiple effects of the same function (updating an argument and returning a value), but independently.
3. If a constructor (setUp) or destructor (tearDown) method fails a corresponding error message is printed and the test is aborted (which will not happen if we used just a local variable of type f_tester allocated on stack).
4. An error message for setUp failure will look like:



5. An error message for teardown failure will look like:



4.3 Writing Unit Tests using Mocks

1. In unit testing we use mocks whenever we need to break down dependencies between a class or function we are writing a test for and between a class or function this *unit under test* depends on. Reasons for doing this may include lack of critical resources (e.g. database or network connectivity), large time required for running unit tests in a real environment (very typical for consumer electronics), and, probably the most important, complexity of test scenario. Even if our *unit under test* depends only on pure computational functions that would deserve representing these functions with mocks just in order to simplify the testing. Indeed, in order to test a function with M possible branches, which depends on another function with N possible branches, we will need to supply M*N test cases in order to provide a required level of test coverage. This grows exponentially and does not scale well. If we use mocks, we will need to supply M+N test cases, and this number grows linearly. For additional useful discussion of the role of mocks read the "Mock Roles, Not Objects" article at http://www.jmock.org/oopsla2004.pdf.
2. As the most of existing automatic mock generators such as EasyMock (www.easymock.org), JMock (www.jmock.org) and NMock (www.nmock.org) the **bmock** library follows a so-called *Record/Replay* paradigm. We first *Record* our test scenario in a form of expectations towards mocks to be used indirectly in the test, and then we *Replay* this test scenario by invoking one or more functions from the unit under test. In its most simple form a unit test using mocks will look like:

```
#include <StdAfx.h>

static BMOCK_VOID_FUNCTION(f,1,(IN(int,x)))

static void g(int x) {
	f(x);
}
struct arg_tester {
	static const int expected1 = 10;
	static const int expected2 = 11;
};

BMOCK_TEST(arg_tester,testArgumentsValidation_Ok) {
	BMOCK_EXPECT(f(expected1));		//record a mock expectation
	BMOCK_REPLAY;				//switch to replay mode
	BOOST_CHECK_NO_THROW(g(expected1));//invoke a function under test
	BMOCK_VERIFY;				//verify expectations fulfillement
}
```

3. For a detailed description of mock specification see the next chapter.
4. Unlike JMock and NMock and similar to EasyMock the **bmock** library in its current form records all expectations via direct invocation of mock functions. Primarily this is done in order to support a strong type checking and a proper handling of functions overloading. On the flip side this approach currently prevents from implementing some advanced types of constrains such as Ignore, Less, Greater, Range, etc. These capabilities might be supported in the future versions of **bmock** .
5. Notice the usage of, the BOOST_CHECK_NO_THROW macro, in the function under test invocation. Using this macro is highly recommended since it guarantees that any failure in the course of the test execution will be properly diagnosed. For tests, which *are expected* to fail use the BOOST_CHECK_THROW macro (see http://www.boost.org/libs/test/doc/index.html)
6. Use BMOCK_REPLAY for switching between Record and Replay modes.
7. At the end of the test use BMOCK_VERIFY macro in order to validate that all expectations have been fulfilled.
8. After the BMOCK_VERIFY macro another Record/Replay cycle could start, though it's recommended to use this advanced feature wisely and only when it's really necessary. Otherwise it's better to write a separate test.

4.4 Specifying Expectations

BMOCK_EXPECT(void_function(arg1,arg2, …))
BMOCK_EXPECT_RETURN (ret_value,function(arg1,arg2,…))
BMOCK_EXPECT_THROW(**new** exception_class(),function(arg1,arg2,…))
BMOCK_EXPECT_CALLBACK(callback_functor)
BMOCK_REPEAT(number_of_times_to_repeat)

in_out_raw_mem(input_value,output_value)
in_out_str(input_C_string,output_C_string)
in_out_raw_mem(input_buffer,input_length,output_buffer,output_length)
out_raw_mem(output_value)
out_str(output_C_string)
out_raw_mem(output_buffer,output_length)
in_raw_mem(input_value)
in_str(input_C_string)
in_raw_mem(input_buffer,input_length)

1. Values for input arguments are specified as is
2. Values for output arguments are specified as is. For example:

```
#include <StdAfx.h>

static BMOCK_FUNCTION(int,f,3,(IN(int,x),OUT(int,&y),OUT(int,*z)))

struct update_arguments {
```

```
        update_arguments()
              :y_expected(y)
              ,z_expected(z)
              ,y_actual(0)
              ,z_actual(0)
              ,r_actual(0)
        {}

        static const int y = 39;
        static const int z = 57;
        static const int x = 25;
        static const int r = 197;

        int y_expected;
        int z_expected;
        int y_actual;
        int z_actual;
        int r_actual;
};

BMOCK_TEST(update_arguments,test_full) {
        BMOCK_EXPECT_RETURN(r,f(x,y_expected,&z_expected));
        BMOCK_REPLAY;
        BOOST_CHECK_NO_THROW(r_actual = f(x,y_actual,&z_actual));
        BMOCK_VERIFY;
        BOOST_CHECK_EQUAL(r,r_actual);
        BOOST_CHECK_EQUAL(y,y_actual);
        BOOST_CHECK_EQUAL(z,z_actual);
}
```

3. Notice, that in the case of OUT arguments what is supplied are values the corresponding arguments should obtain. Multiple settings are freely supported, for example (suing the same fixture as above):

```
static BMOCK_VOID_FUNCTION(g,2,(OUT(int,&y),OUT(int,*z)))
BMOCK_TEST(update_arguments,test_multiple) {
        BMOCK_EXPECT(g(y_expected,&z_expected));
        ++y_expected;
        ++z_expected;
        BMOCK_EXPECT(g(y_expected,&z_expected));
        BMOCK_REPLAY;
        BOOST_CHECK_NO_THROW(g(y_actual,&z_actual));
        BOOST_CHECK_EQUAL(y,y_actual);
        BOOST_CHECK_EQUAL(z,z_actual);
        BOOST_CHECK_NO_THROW(g(y_actual,&z_actual));
        BOOST_CHECK_EQUAL(y+1,y_actual);
        BOOST_CHECK_EQUAL(z+1,z_actual);
        BMOCK_VERIFY;
}
```

4. Values for input/output arguments are supplied using a special **`in_out_raw_mem`** function, for example:

```
#include <StdAfx.h>
using namespace bmock;
```

```
static BMOCK_FUNCTION(int,f,3,(IN(int,x)
                               ,IN_OUT(int &,y),IN_OUT(int,*z)))

struct in_out_arguments {
      in_out_arguments()
            :y_actual(y_in)
            ,z_actual(z_in)
            ,r_actual(0)
      {
            BMOCK_EXPECT_RETURN(r,f(x,*in_out_raw_mem(y_in,y_out)
                                    ,in_out_raw_mem(z_in,z_out)));
            BMOCK_REPLAY;
            BOOST_CHECK_NO_THROW(r_actual = f(x,y_actual,&z_actual));
            BMOCK_VERIFY;
      }
      static const int x     = 25;
      static const int y_in  = 39;
      static const int y_out = 139;
      static const int z_in  = 57;
      static const int z_out = 157;
      static const int r     = 197;

      int              y_actual;
      int              z_actual;
      int              r_actual;
};

BMOCK_TEST(in_out_arguments,test_return_value) {
      BOOST_CHECK_EQUAL(r,r_actual);
}

BMOCK_TEST(in_out_arguments,test_reference_argument) {
      BOOST_CHECK_EQUAL(y_out,y_actual);
}

BMOCK_TEST(in_out_arguments,test_pointer_argument) {
      BOOST_CHECK_EQUAL(z_out,z_actual);
}
```

5. Notice another possible (and actually classical!) use of fixture.
6. Sometimes even though an argument is specific as input/output all, what is required in a particular test, is its output value. This could happen when an input value has no meaning for a particular function call (often happens with many system level APIs), for some reason cannot be predicted by the test, or just is not interesting for this particular test. This kind of input/output arguments are sometimes called input/output-output arguments. To specify properly these arguments a special **out_raw_mem** function is used. For example (using the same functions):

```
struct out_arguments {
      out_arguments()
            :y_actual(0)//still need to initialized
            ,z_actual(0)
            ,r_actual(0)
      {
            BMOCK_EXPECT_RETURN(r,f(x,*out_raw_mem(y_out)
```

```
                                                ,out_raw_mem(z_out)));
            BMOCK_REPLAY;
            BOOST_CHECK_NO_THROW(r_actual = f(x,y_actual,&z_actual));
            BMOCK_VERIFY;
      }
      static const int x     = 25;
      static const int y_out = 139;
      static const int z_out = 157;
      static const int r     = 197;

      int              y_actual;
      int              z_actual;
      int              r_actual;
};

BMOCK_TEST(out_arguments,test_all) {
      BOOST_CHECK_EQUAL(r,r_actual);
      BOOST_CHECK_EQUAL(y_out,y_actual);
      BOOST_CHECK_EQUAL(z_out,z_actual);
}
```

7. Sometimes in a certain test scenario input/output arguments are actually treated as input-only. For that purpose a special **in_raw_mem** function is used. For example (using the same functions as above):

```
struct in_arguments {
      in_arguments()
            :y_actual(y_in)
            ,z_actual(z_in)
            ,r_actual(0)
      {
            BMOCK_EXPECT_RETURN(r,f(x,*in_raw_mem(y_in)
                                    ,in_raw_mem(z_in)));
            BMOCK_REPLAY;
            BOOST_CHECK_NO_THROW(r_actual = f(x,y_actual,&z_actual));
            BMOCK_VERIFY;
      }
      static const int x     = 25;
      static const int y_in  = 39;
      static const int z_in  = 57;
      static const int r     = 197;

      int              y_actual;
      int              z_actual;
      int              r_actual;
};

BMOCK_TEST(in_arguments,test_all) {
      BOOST_CHECK_EQUAL(r,r_actual);
      BOOST_CHECK_EQUAL(y_in,y_actual);//not changed!
      BOOST_CHECK_EQUAL(z_in,z_actual);//not changed!
}
```

8. Input/output arguments, which are null-terminated strings, need to be treated in a special way. For that purpose special **in_out_str**, **out_str**, and **in_str** functions have to be used. For example:

```
#include <StdAfx.h>
using namespace bmock;

static BMOCK_VOID_FUNCTION(h,1,(IN_OUT(char *const,str)))

static const char s_str[]     = "qwertyuiop[]";
static char       s_str_exp[] = "asdfghjkl;'zxcvbnm,.";

struct in_out_string_argument {
      in_out_string_argument()
      {
            strcpy(buf,s_str);
      }

      static const size_t L = sizeof(s_str_exp);
      char                buf[L];
};

BMOCK_TEST(in_out_string_argument,test_char_OK) {
      BMOCK_EXPECT(h(in_out_str(s_str,s_str_exp)));
      BMOCK_REPLAY;
      BOOST_CHECK_NO_THROW(h(buf));
      BMOCK_VERIFY;
      BOOST_CHECK_EQUAL(mismatch_at(buf,buf+L,s_str_exp),L);
}
```

9. There is a certain tradeoff between using null-terminated strings and RAW_MEM buffers. The former are slightly more convenient since there the arguments length is calculated automatically. The latter is safer, especially for OUT and IN_OUT arguments, since in this case maximal output length restriction is validated.
10. By default **unsigned char** * pointers are treated as null-terminated strings. The main reason for this decision was the fact that many legacy “C” projects use **unsigned char*** pointers for null-terminated strings.
11. The **mismatch_at** function is used to comparing the expected and actual strings such that it would be possible to report an exact first position where the two strings differ if any.
12. Values of output or input/output arguments, which are arrays or non-typed raw memory buffers, have to be supplied using a full form of the **in_out_raw_mem**, **out_raw_mem**, and **in_raw_mem** functions accordingly (notice that for values of input arguments only the buffer needs to be supplied one should **not** supply these values using **in_raw_mem**). For example:

```
#include <StdAfx.h>
#include <algorithm>
#include <boost/lambda/lambda.hpp>
using namespace bmock;

static BMOCK_VOID_FUNCTION(g,2,(
```

```
                        RAW_MEM(OUT,void *const,outbuf,len)
                        ,IN(const size_t,len)))
static BMOCK_VOID_FUNCTION(h,2,(
                        RAW_MEM(IN_OUT,void *const,buf,len)
                        ,IN(const size_t,len)))

static const unsigned char inp_buf[] = {0x00,0x01,0x02,0x03
                                        ,0x04,0x05,0x06,0x07
                                        ,0x08,0x09,0x0A,0x0B
                                        ,0x0C,0x0D,0x0E,0x0F};
static const size_t        L1 = sizeof(inp_buf);
static const unsigned char out_buf[] = {0x10,0x11,0x12,0x13
                                        ,0x14,0x15,0x16,0x17
                                        ,0x18,0x19,0x1A,0x1B
                                        ,0x1C,0x1D,0x1E,0x1F};
static const size_t        L2 = sizeof(out_buf);

static const size_t        DELTA = 10;
static const size_t        L3    = L2+DELTA;
static unsigned char       act_buf[L3];

struct out_raw_buffer {
      out_raw_buffer() {
            std::fill_n(act_buf,L3,0);
      }

      static const size_t  DELTA = 10;
      static const size_t  L3    = L2+DELTA;
      unsigned char        act_buf[L3];
};


BMOCK_TEST(out_raw_buffer,test_OK) {
      using namespace boost::lambda;
      BMOCK_EXPECT(g(out_raw_mem(out_buf,L2),L3));
      BMOCK_REPLAY;
      BOOST_CHECK_NO_THROW(g(act_buf,L3));
      BMOCK_VERIFY;
      BOOST_CHECK_EQUAL(L2,mismatch_at(out_buf,out_buf+L2,act_buf));
      BOOST_CHECK_EQUAL(DELTA,index_of(act_buf+L2,act_buf+L3,_1!=0));
}

struct in_out_raw_buffer : public out_raw_buffer {
      in_out_raw_buffer()
      {
            std::copy(inp_buf,inp_buf+L1,act_buf);
      }
};

BMOCK_TEST(in_out_raw_buffer,test_in_out_raw_buffer_OK) {
      using namespace boost::lambda;
      BMOCK_EXPECT(h(in_out_raw_mem(inp_buf,L1,out_buf,L2),L1));
      BMOCK_REPLAY;
      BOOST_CHECK_NO_THROW(h(act_buf,L1));
      BMOCK_VERIFY;
      BOOST_CHECK_EQUAL(L2,mismatch_at(out_buf,out_buf+L2,act_buf));
      BOOST_CHECK_EQUAL(DELTA,index_of(act_buf+L2,act_buf+L3,_1!=0));
```

```
}
```

13. In the last test the **index_of** function is used in order to validate that all bytes of the **act_buf** are non-zero. For all kind of pointer arguments a NULL value supplied in expectation is treated as is, that means it's expected to obtain the NULL pointer for this argument (IN, OUT or IN_OUT). However within IN_OUT argument expectation pair the NULL pointer is treated as ignore. For example in_out_raw_mem(NULL,0,NULL,0) would mean "ignore the argument".
14. For any argument you want to ignore you need to use the IGN(type,name) macro. for example:

```
#include <StdAfx.h>
using namespace bmock;
using namespace boost;

static const int           actual       = 10;

static BMOCK_VOID_FUNCTION(ig,2,(IGN(int,x),IN(int,y)))

BOOST_AUTO_UNIT_TEST(testArgumentsValidationIGNMacrodifferentValue) {
      int x=12;
      BMOCK_CONTROL(c);
      BMOCK_EXPECT_C(c,ig(actual,x));
      BMOCK_REPLAY_C(c);
      BOOST_CHECK_NO_THROW(ig(x,x));
      BMOCK_VERIFY_C(c);
}

BOOST_AUTO_UNIT_TEST(testArgumentsValidationIGNMacroSameValue) {
      int x=10;
      BMOCK_CONTROL(c);
      BMOCK_EXPECT_C(c,ig(actual,x));
      BMOCK_REPLAY_C(c);
      BOOST_CHECK_NO_THROW(ig(x,x));
      BMOCK_VERIFY_C(c);

}
```

15. Callback expectations are provided for the most recent BMOCK_EXPECT, BMOCK_EXPECT_RETURN, BMOCK_EXPECT_THROW, BMOCK_STUB, BMOCK_STUB_RETURN or BMOCK_STUB_THROW. For example:

```
#include <StdAfx.h>
#include <boost/bind.hpp>
#include <cstring>
using namespace bmock;
using namespace boost;

static BMOCK_VOID_FUNCTION(f,1,(IN(const char *const,str)))
static BMOCK_VOID_FUNCTION(g,1,(OUT(char *const,str)))

static const char s_str[] = "qwertyuiop[]";

struct callback_tester {
      callback_tester() {
```

```
            std::fill_n(buf,sizeof(buf),0);
            BMOCK_EXPECT(f(s_str));
            BMOCK_EXPECT_CALLBACK(bind(g,buf));
            BMOCK_EXPECT(g((char *)s_str));
            BMOCK_REPLAY;
      }
      ~callback_tester() {
            BMOCK_VERIFY;
      }
      static const size_t L = sizeof(s_str);
      char buf[L];
};

BMOCK_TEST(callback_tester,test_OK) {
      BOOST_CHECK_NO_THROW(f(s_str));
      BOOST_CHECK_EQUAL(mismatch_at(buf,buf+L,s_str),L);
}

struct callback {
      callback() {
            std::fill_n(buf,sizeof(buf),0);
      }

      static const size_t L = sizeof(s_str);
      char buf[L];
      int x;
};
BMOCK_TEST(callback,test_callback_invoked_by_last_stub) {
      BMOCK_EXPECT(g((char *)s_str));
      BMOCK_EXPECT(f(s_str));
      BMOCK_STUB(k(x));
      BMOCK_EXPECT_CALLBACK(bind(g,buf));
      BMOCK_REPLAY;
      BOOST_CHECK_NO_THROW(k(x));
      BOOST_CHECK_NO_THROW(f(s_str));
      BOOST_CHECK_EQUAL(L,mismatch_at(buf,buf+L,s_str));
      BMOCK_VERIFY;
}
```

16. Usually callback functors are constructed using the Boost library **lambda::bind** function. Using a stand-alone version of the **bind** function is not recommended due to potential conflicts with some **bmock** library headers.
17. More then one callback can be specified.
18. In the current version of the **bmock** library callbacks are invoked within the scope of the most recent mock function right after the input arguments validation.
19. Repeat expectations are provided for the most recent BMOCK_EXPECT, BMOCK_EXPECT_RETURN or BMOCK_EXPECT_THROW. For example:

```
#include <StdAfx.h>
using namespace bmock;
using namespace boost;
using namespace std;

static BMOCK_VOID_FUNCTION(f,0,())
struct repetition_tester{
      int result;
```

```
};

BMOCK_TEST(repetition_tester,testVoidFunctionCall_Repetition) {
	BMOCK_EXPECT(f());
	BMOCK_REPEAT(5);
	BMOCK_REPLAY;
	for(int i=0;i<5;i++)
		BOOST_CHECK_NO_THROW(f());
	BMOCK_VERIFY;
}
```

4.5 Specifying Expectations for Constructor and Destructor

1. Expectations for regular C++ class member functions are specified in the same way. Currently expectations for **this** pointer are not supported. This limitation might be reconsidered in the future.
2. In order to specify expectations for constructor and destructor some extra effort is required. For example:

```
#include <stdafx.h>

struct test_mock_class {
	test_mock_class(int x,int *y);
	~test_mock_class();
};

BMOCK_CONSTRUCTOR(test_mock_class,2,(IN(int,x),OUT(int *,y)))
BMOCK_DESTRUCTOR(test_mock_class);

struct test_mock_class_fixture {
	test_mock_class_fixture()
		:x(10)
		,exp_y(11)
		,y(0)
		,ptr(NULL)
	{}

	const int		x;
	int			exp_y;
	int			y;
	test_mock_class *ptr;
};

BMOCK_TEST(test_mock_class_fixture,test_contructor_destructor) {
	BMOCK_EXPECT(ptr = new test_mock_class(x,&exp_y));
	BMOCK_EXPECT(delete ptr);
	BMOCK_REPLAY;
	BOOST_CHECK_NO_THROW({ test_mock_class tst(x,&y);});
	BMOCK_VERIFY;
	BOOST_CHECK_EQUAL(exp_y,y);
}

BMOCK_TEST(test_mock_class_fixture,test_contructor_only) {
	test_mock_class *ptr1;

	BMOCK_EXPECT(ptr = new test_mock_class(x,&exp_y));
```

```
	BMOCK_REPLAY;
	BOOST_CHECK_NO_THROW(ptr1 = new test_mock_class(x,&y));
	BMOCK_VERIFY;
	BOOST_CHECK_EQUAL(exp_y,y);
	//tierDown
	BMOCK_EXPECT(delete ptr);
	BMOCK_REPLAY;
	delete ptr1;
	BMOCK_VERIFY;
}

BMOCK_TEST(test_mock_class_fixture,test_destructor_only) {
	//setUp
	test_mock_class *ptr1;

	BMOCK_EXPECT(ptr = new test_mock_class(x,&exp_y));
	BMOCK_REPLAY;
	BOOST_CHECK_NO_THROW(ptr1 = new test_mock_class(x,&y));
	BMOCK_VERIFY;
	//test
	BMOCK_EXPECT(delete ptr);
	BMOCK_REPLAY;
	BOOST_CHECK_NO_THROW(delete ptr1;);
	BMOCK_VERIFY;
}
```

3. In order to specify an expectation for a constructor some object needs to be created in the Record mode. For example:

```
BMOCK_EXPECT(ptr = new test_mock_class(x,&exp_y));
```

4. In order to specify an expectation for a destructor a previously created object needs to be destroyed. For example:

```
BMOCK_EXPECT(delete ptr);
```

5. Sometimes it might look a bit unusual and require an additional EXPECT-REPLAY-VERIFY cycle. For example:

```
BMOCK_TEST(test_mock_class_fixture,test_destructor_only) {
	//setUp
	test_mock_class *ptr1;

	BMOCK_EXPECT(ptr = new test_mock_class(x,&exp_y));
	BMOCK_REPLAY;
	BOOST_CHECK_NO_THROW(ptr1 = new test_mock_class(x,&y));
	BMOCK_VERIFY;
	//test
	BMOCK_EXPECT(delete ptr);
	BMOCK_REPLAY;
	BOOST_CHECK_NO_THROW(delete ptr1;);
	BMOCK_VERIFY;
}
```

4.6 Specifying Stub Expectations

BMOCK_STUB(void_function(arg1,arg2, …))
BMOCK_STUB_RETURN (ret_value,function(arg1,arg2,…))
BMOCK_STUB_THROW(**new** exception_class(),function(arg1,arg2,…))

1. Stubs can be called any number of times (or not at all), and it does not matter when they are called.
2. The IN arguments of stubs are ignored, and the OUT arguments are treated as OUT arguments for expectations.
3. For each function one can specify only one stub.

For example:

```
BMOCK_VOID_FUNCTION(g,0,())
BMOCK_VOID_FUNCTION(h,2,(IN(int,x),OUT(int&,y)))
BMOCK_FUNCTION(int,f,2,(IN(int,x),OUT(int&,y)))

BMOCK_TEST(arg_tester,test_void_stub_no_arguments_with_other_expectatio
n_multiple_calls_to_stub)
{
      int x=4;
      int y=8;
      int z_1=0;
      int z_2=0;
      int r=0;
      BMOCK_EXPECT_RETURN(10,f(x,y));
      BMOCK_STUB(g());
      BMOCK_EXPECT(h(x,y));
      BMOCK_REPLAY;
      BOOST_CHECK_NO_THROW(g());
      BOOST_CHECK_NO_THROW(r=f(x,z_1));
      BOOST_CHECK_NO_THROW(g());
      BOOST_CHECK_NO_THROW(h(x,z_2));
      BOOST_CHECK_NO_THROW(g());
      BMOCK_VERIFY;
      BOOST_CHECK_EQUAL(y,z_1);
      BOOST_CHECK_EQUAL(y,z_2);
}


BMOCK_TEST(arg_tester,test_void_stub_no_arguments_with_other_expectatio
n_no_calls_to_stub)
{
      int x=4;
      int y=8;
      int z_1=0;
      int z_2=0;
      int r=0;
      BMOCK_EXPECT_RETURN(10,f(x,y));
      BMOCK_STUB(g());
      BMOCK_EXPECT(h(x,y));
      BMOCK_REPLAY;
      BOOST_CHECK_NO_THROW(r=f(x,z_1));
      BOOST_CHECK_NO_THROW(h(x,z_2));
      BMOCK_VERIFY;
```

```
	BOOST_CHECK_EQUAL(y,z_1);
	BOOST_CHECK_EQUAL(y,z_2);
}
```

4. If there is an expectation and a stub for the same function the expectation overwrites the stub:

```
BMOCK_TEST(arg_tester,test_void_stub_with_arguments_with_other_expectat
ion_multiple_calls_to_stub_expectation_overwrites_stub)
{
	int x=4;
	int y=8;
	int x_1=3;
	int y_1=9;
	int z_1=0;
	int z_2=0;
	int l_1=0;
	int l_2=0;
	BMOCK_EXPECT_RETURN(10,f(x,y));
	BMOCK_STUB(h(x_1,y_1));
	BMOCK_EXPECT(h(x,y));
	BMOCK_REPLAY;
	BOOST_CHECK_NO_THROW(h(x,l_1));
	BOOST_CHECK_NO_THROW(r=f(x,z_1));
	BOOST_CHECK_NO_THROW(h(x,z_2));
	BOOST_CHECK_NO_THROW(h(x,l_2));
	BMOCK_VERIFY;
	BOOST_CHECK_EQUAL(y,z_1);
	BOOST_CHECK_EQUAL(y,z_2);
	BOOST_CHECK_EQUAL(y_1,l_1);
	BOOST_CHECK_EQUAL(y_1,l_2);
}
```

5 Specifying Mocks

There are two ways to specify mocks:

- Not Dynamic Mocks- who are specified in a different project than the real function.
- Dynamic Mocks- who are specified with the real function's declaration and definition.

5.1 Specifying Not Dynamic Mocks

BMOCK_VOID_FUNCTION(function_name,#of_arguments,(argument_list))
BMOCK_FUNCTION(return_type, function_name,#of_arguments,(argument_list))
BMOCK_CONSTRUCTOR(class_name,#of_arguments,(argument_list))
BMOCK_DESTRUCTOR(class_name);
BMOCK_METHOD(ret_type, class_name, method_name,#of_arguments,(arg_list))
BMOCK_VOID_METHOD(class_name, method_name,#of_arguments,(arg_list))
BMOCK_CONST_METHOD(ret_type, class_name, method,#of_arguments,(arg_list))
BMOCK_CONST_VOID_METHOD(class_name, method,#of_arguments,(arg_list))

IN(type,name)
OUT(type,name)
IN_OUT(type,name)
IN_PTR(type,name)
IN_OUT_PTR(type,name)
RAW_MEM(IN,type,name,input_length)
RAW_MEM(OUT,type,name,max_output_length)
RAW_MEM(IN_OUT,type,name,max_output_length)
CLBK(type,name)
IGN(type,name)

1. In order to use these kind of mocks BMOCK_GENERATE_CODE should not be defined.
2. Mock functions are specified with an IDL-like macro annotation.
3. Built-in and string (including **unsigned char *** pointers) types input arguments are specified using the IN(type, name) macro.
4. Built-in and string (including **unsigned char *** pointers) types output arguments are specified using the OUT(type, name) macro.
5. Built-in and string (including **unsigned char *** pointers) types input/output arguments are specified using the IN_OUT(type, name) macro.
6. Pointer (void, partial types) input arguments are specified using the IN_PTR(type, name) macro.
7. Function pointers input arguments are specified using the CLBK(type, name) macro.
8. Pointer (void, functions or partial types) input/output arguments are specified using the IN_OUT_PTR(type, name) macro.
9. Raw-memory input arguments are specified using the RAW_MEM(IN,type,name,input_length) macro.
10. Raw-memory output arguments are specified using the RAW_MEM(OUT,type,name,max_output_length) macro.
11. Raw-memory input/output arguments are specified using the RAW_MEM(IN_OUT,type,name,max_output_length) macro.
12. In order to ignore the argument's value use the IGN(type,name) macro.
13. For any argument with a non-built-in type you will need to decide about its serialization mechanism. It could either be a hexadecimal dump or a formatted output. If it's hexadecimal dump you will need to declare it as a RAW_MEM with length 1. If it's a formatted output will need to define for it << (shift left) and >> (shift right) C++ operators within the std:: namespace. For example:

```
#include <StdAfx.h>
#include <iostream>
#include <semaphore.h>

namespace std {
      ostream &operator <<(ostream &os,const sem_t &s) {
            os << s.sem << ' ' << s.nbWaiters;
            return os;
      }
```

```
	istream &operator >>(istream &is,sem_t &s) {
		char blank;
		is >> s.sem;
		is >> blank;
		is >> s.nbWaiters;
		return is;
	}
}
```

14. Enum serialization is supported in test mode. For details on console mode support see BMock Console Mode chapter.

15. If a data structure has internal pointers (e,g, char *) to be dynamically allocated during de-serialization, the programmer can use the function bmock::get_buffer(ptr, length) which allocates memory and takes care of the memory cleanup (void* are treated as unsigned chars*) . To be more accurate this problem exists only for plain “C” structures, which do not have destructors. For example:

```
struct serialization_test_data2 {
	serialization_test_data2()
		:data(NULL)
		,length(0)
	{}
	int* data;
	char* Str;
	void* data_;

};

namespace std {

	inline ostream &operator <<(ostream &os,const
serialization_test_data2 &td) {
		os<<*td.data;
		os<<' ';
		os<<td.Str;
		os<<' ';
		os<<(unsigned char*)td.data_;
		return os;
	}
	inline istream &operator >>(istream &is,serialization_test_data2
&td) {
		char blank;
		bmock::get_buffer(td.data,1);
		is>>*td.data;
		is>>blank;
		bmock::get_buffer(td.Str,9);
		is>>td.Str;
		is>>blank;
		bmock::get_buffer(td.data_,36);
		is>>(unsigned char*)td.data_;
		return is;
	}
```

}

5.2 Specifying Dynamic Mocks

1. Dynamic Mocks can be declared only if BMOCK_GENERATE_CODE is defined. And in this mode they are the only type of mocks that can be specified.
2. In order to generate mocks BMOCK_USE_MOCKS should be defined. If (for example for release purpose) one wants to build all the function as regular functions one only needs to not define BMOCK_USE_MOCKS.
3. A dynamic mock for a function/method or void function is specified this way:

   BMOCK_SPECIFIC_MACRO
   Implementation of the function
   BMOCK_END

   For example:

   BMOCK_VOID_FUNCTION(f,0,())
   x=0;
   BMOCK_EN

   BMOCK_FUNCTION(int,function_k,0,())
   return 0;
   BMOCK_END

   BMOCK_METHOD(int,test_generate_mock_class,h,1,(IN(int,x)))
   my_val=x
   ;return my_val;
   BMOCK_END

   BMOCK_CONSTRUCTOR(test_generate_mock_class,1,(OUT(double&,x)))
   my_val=24;
   x=my_val;
   BMOCK_END

   The bmock function declaration macros are the same for dynamic and not dynamic mocks.

4. In order that a BMOCK_TEST will generate a mock for a function declared as a dynamic mock the macro BMOCK_CREATE_FUNCTION_MOCK should be used at the beginning of the test.
5. BMOCK_CREATE_FUNCTION_MOCK can recive as an argument :
   - The function name BMOCK_CREATE_FUNCTION_MOCK(f);-the test will generate a mock for f.
   - A module name followed by a * BMOCK_CREATE_FUNCTION_MOCK(C_MODULE_NAME*);-the test will generate a mock for all functions with C_MODULE_NAME at the

beginning of their name(c_MODULENAME_F etc.) (that were declared as dynamic mocks).

- The function declaration (with or without the argument names) inside quotation marks.
  BMOCK_CREATE_FUNCTION_MOCK("void f_1(int)");
  BMOCK_CREATE_FUNCTION_MOCK("int* f(int *,char &, string)");- the test will generate a mock for the specified function only (not for the other overloads).

6. In order that a BMOCK_TEST will generate a mock for a method declared as a dynamic mock the macros BMOCK_CREATE_METHOD_MOCK should be used at the beginning of the test.
7. BMOCK_CREATE_METHOD_MOCK can recive as an argument :

   - A class name BMOCK_CREATE_METHOD_MOCK(class_name);-the test will generate a mock for all methods of the class (that were declared as dynamic mocks).
   - BMOCK_CREATE_METHOD_MOCK(class_name::my_method);-the test will generate a mock for the method my_method.
   - BMOCK_CREATE_METHOD_MOCK(class_name::method_name_prefix*); -the test will generate a mock for all the methods of the class that have this prefix.
   - The method declaration (with or without the argument names) inside quotation marks.
     BMOCK_CREATE_METHOD_MOCK("void class::method_f_1(int)");
     BMOCK_CREATE_METHOD_MOCK("int* class::method_f(int *,char &, string)");- the test will generate a mock for the specified method only (not for the other overloads).

8. The functions that the test did not generate a mock for will be treated as regular functions.
9. In order that the code would compile without the boost library one needs to include some boost files. The directory Boost\boost_for_production_code holds all the necessary files.

5.3 Specifying Mocks for COM Objects

1. At a bottom line Microsoft COM objects are translated into C++ abstract classes with a set of pure virtual functions. Therefore for any COM object in principle its mock version could be created.
2. In order to create a mock version for some COM object a new class has to be derived from the original COM object's interface redefining ALL its methods including those, which are defined for the IUnknown interface. For example:

```
#pragma once
#include <comdef.h>
```

```
#include <initguid.h>
#include <imceprop.h>


class CMock_IMCMpegVideoEncoder :
      public IMCMpegVideoEncoder
{
public:
    STDMETHOD(get_VideoMpegType)(THIS_ LPDWORD lpdwMpegType);
    STDMETHOD(put_VideoMpegType)(THIS_ DWORD   dwMpegType);
//other methods of IMCMpegVideoEncoder
    HRESULT STDMETHODCALLTYPE QueryInterface(
             REFIID riid,void __RPC_FAR *__RPC_FAR *ppvObject);
    ULONG STDMETHODCALLTYPE AddRef( void);
    ULONG STDMETHODCALLTYPE Release( void);
};
```

3. Then in a separate .cpp file a mock version for each method has to be specified. For example:

```
#include <stdafx.h>
#include "mock_mpeg_dlg.h"

BMOCK_FUNCTION(HRESULT,
      CMock_IMCMpegVideoEncoder::GetDefaultVideoSettings,
      3,(
      IN(DWORD,dwMpegType),
      IN(DWORD,dwVideoMode),
      OUT(mpeg_v_settings*, pVSettings)
))

BMOCK_FUNCTION(HRESULT,
      CMock_IMCMpegVideoEncoder::put_VideoSettings,
      1,
      (IN(mpeg_v_settings*, pVSettings)
))
```

4. Specifying a constructor or a destructor for this kind of mock classes is usually not required.

6 Using BMock Console Mode

1. The BMockConsoleApp project template is intended to support integration with Java version of the FitLibrary using the same definition of mock functions.
2. In this mode expectations are not set and the **bmock** library prints names of all functions and values of their input arguments into standard output stream. In addition it reads values of output arguments from the standard input stream.
3. The BMockConsoleApp project template uses the Boost.Program_options library in order to ensure a consistent handling of all command line arguments. In order to add declaration and processing of your unit-specific command line arguments modify accordingly the main.cpp file automatically created by the BMockConsoleApp project wizard:

```
#include <stdafx.h>

int cpp_main(int argc, char * argv[]) {
      bmock::options opts("<put your application name here>");
      opts.add_options()
            //put your application options here
            //(see http://www.boost.org/doc/html/program_options.html)
            //for format specification
      ;
      opts.parse(argc,argv);
      //run your application here
      return 0;
}
```

4. The format specification for adding application options to bmock::options is as follows:

```
bmock::options opts("<put your application name here>");
opts.add_options()
      ("custom_option1","boolean option")
      ("custom_option2",value<int>(),"integer option")
      ("custom_option3",value<int>()->default_value(-1),"another
integer option")
      ;
```

The first parameter is the option name, the second is information about the value (if there is a value), the third is the option's discription. In this example the first option has no value the second has value of type int and the third has value of type int with default value -1.

For more options see http://www.boost.org/doc/html/program_options.html

5. The function `parse(argc,argv)`must be called so that the options defined by the command line will be saved.
6. bmock::options has also a
   `template<typename T> T get(const std::string &opt_name)`function.
   When T is a bool it returns whether the option was defined or not.
   If the option has a value, T can be the value type, and the output of the function will be the value defined for the option.
7. Bmock::option's function:
   `bool defined(const std::string &opt_name)`
   is the same as get < bool>.
8. All BMockConsoleApp handle a special –no_prompt switch. When this switch is supplied via command line, all mock argument names will be printed on a separate line (more convenient for integration with the Fit Library). If not supplied, output argument names and values will be printed on the same line separated by the "=" prompt character (more convenient for interactive mode).
9. In addition a –input=<file_name> switch is supported. When this switch is supplied all values of output arguments are read from the specified file. When an end of the input file is reached values of additional output arguments will be read from the standard input. This feature allows a lightweight automation of running complex testing scenario when it would be more convenient to pass through some initial sequence of mocked function calls automatically.

10. The BMockConsoleApp template automatically configures the –no_prompt and the –input=input.dat command line. It also adds an empty input.dat file to the project.
11. The **bmock** console mode provides a built-in support for multi-threading. That is if two or more threads are trying to write to the console pipe they will be serialized through a mutex semaphore embedded within the console pipe infrastructure.
12. Any input line starting with a **#** (dash) character is treated as a comment and ignored. In order to put a real # dash symbol as the first character put it twice: "##".
13. Another option is to use the FIT log file of a previous run as the input (bmock recognizes that this is a log file by the log extension). This option uses only the no prompt switch. This option is run under debug mode and initializes a debug breakpoint if there is a difference between this run and the previous run described by the log file.
14. By running with the log file one can insert a breakpoint in the run by inserting a "@" at the beginning of the line in the log file where he wants to break. In order to put a real @ symbol as the first character of a line, put it twice: "@@".
15. In console mode bmock allocates memory for return pointers and double pointers out arguments. The default behavior of bmock is to also take care of their deletion, in order to designate that the pointer is deleted externally by the client's function/method and thus bmock does no need to take care of its deletion one should specify the function or out argument as:

    BMOCK_ED_FUNCTION(console_custom_type *,k,0,());
    ED_OUT(console_custom_type **,ptr))
    IN_OUT_ED(console_custom_type **,p_x))
16. The default behavior of bmock when specifying a function, whose return value is a void pointer or has an OUT argument which is a double void pointer, is to read a numeric value. If the pointer is going to be de-referenced one needs to use the macros:

    BMOCK_ALLOC_FUNCTION(void*,f_alloc,0,());
    ALLOC_OUT(void** ,x)
    IN_OUT_ALLOC(void** ,x)

    If the pointers are externally deleted one should use the "ED" macros as described above.
17. In order to specify a function, whose return value is a pointer to an incomplete type or has an OUT argument which is a double pointer to an incomplete type, one needs to use these macros:

    BMOCK_ICP_FUNCTION(incomp*,f_incomp,0,());
    ICP_OUT(incomp** ,x);
    IN_OUT_ICP(incomp** ,x))
18. When specifying a function, whose return value is a function pointer or has an OUT argument which is a double function pointer, one can choose from two options:
    - Specify through the input stream the correct numeric value of the function pointer.
    - If the numeric value is unknown one should specify it with the macro `BMOCK_REGISTER` the function pointer before the beginning of the test run. `BMOCK_REGISTER_METHOD` macro should be used to declare methods. This macro works with either static or non static member functions.

for example:

```
int foo1(){int x=0;return x;}

BMOCK_REGISTER(foo1);

class testclass {
   void f();
}

BMOCK_REGISTER_METHOD(testclass, f);
```

If the function pointer is of a mock function one can specify the function in the mock declartion like this:

```
static BMOCK_ENTRY(FUNCTION(int,foo,0,()),foo).
```

Now when specifying through the input stream the function name, bmock knows how to convert it to the correct numeric value.

19. Enum serialization in console mode is partially supported. Mocks can receive enums as IN arguments but only int value will be printed to the console. For OUT arguments and return values the console will expect the int value of the enum.
20. If one wants to use a dynamic mock as a mock he should declare that using the macro's BMOCK_CREATE_METHOD_MOCK and BMOCK_CREATE_FUNCTION_MOCK inside the cpp_main. (the macro's are used as explained in the dynamic mocks chapter)

6.1 Integration with the FIT Library

1. The **bmock** library comes with a built-in support for integration between console mode and popular Framework for Integrated Tests (FIT) library http://sourceforge.net/projects/fitlibrary. The FIT library is mostly useful for writing acceptance tests.
2. Acceptance tests are prepared in a form of HTML tables. For example:

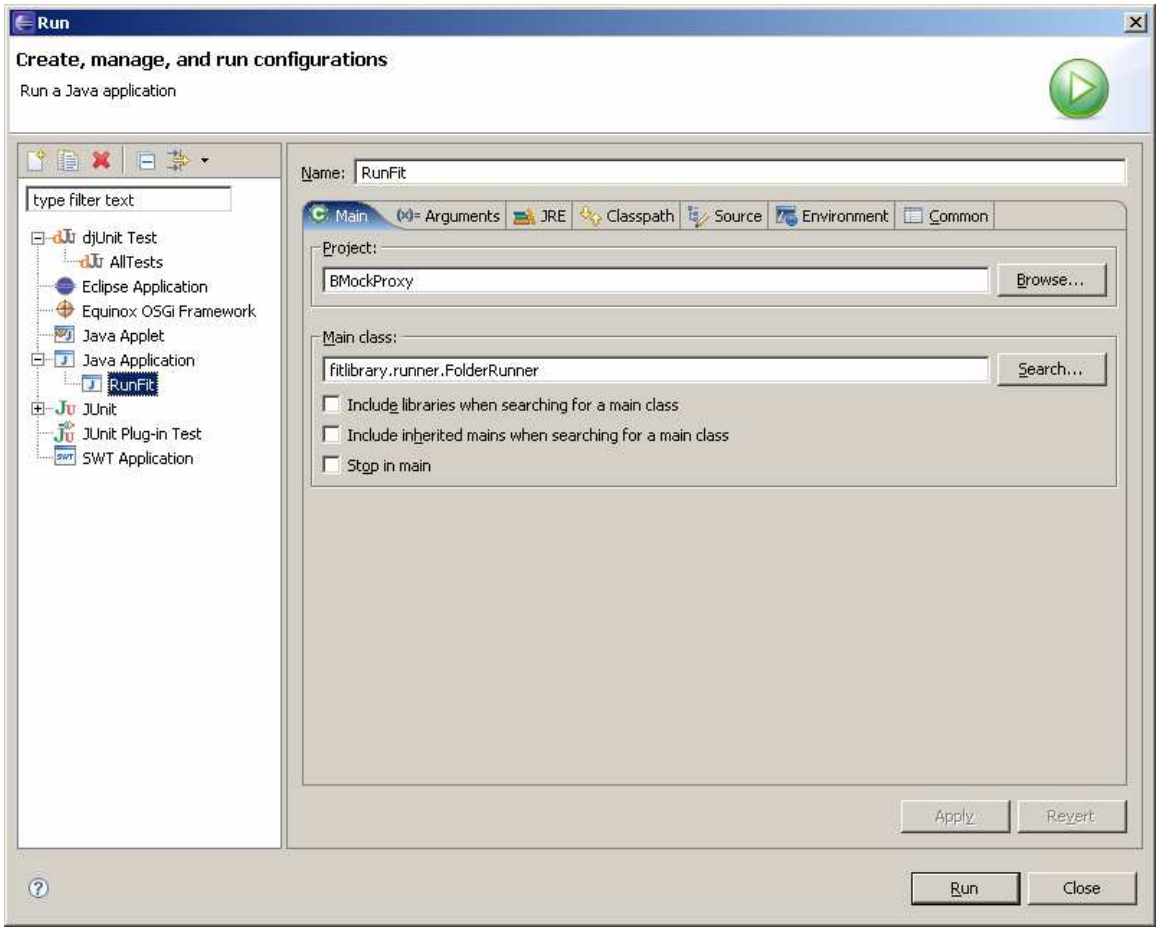
Sample Event Queue Consumer

com.nds.bmock.console.SampleQueueConsumer

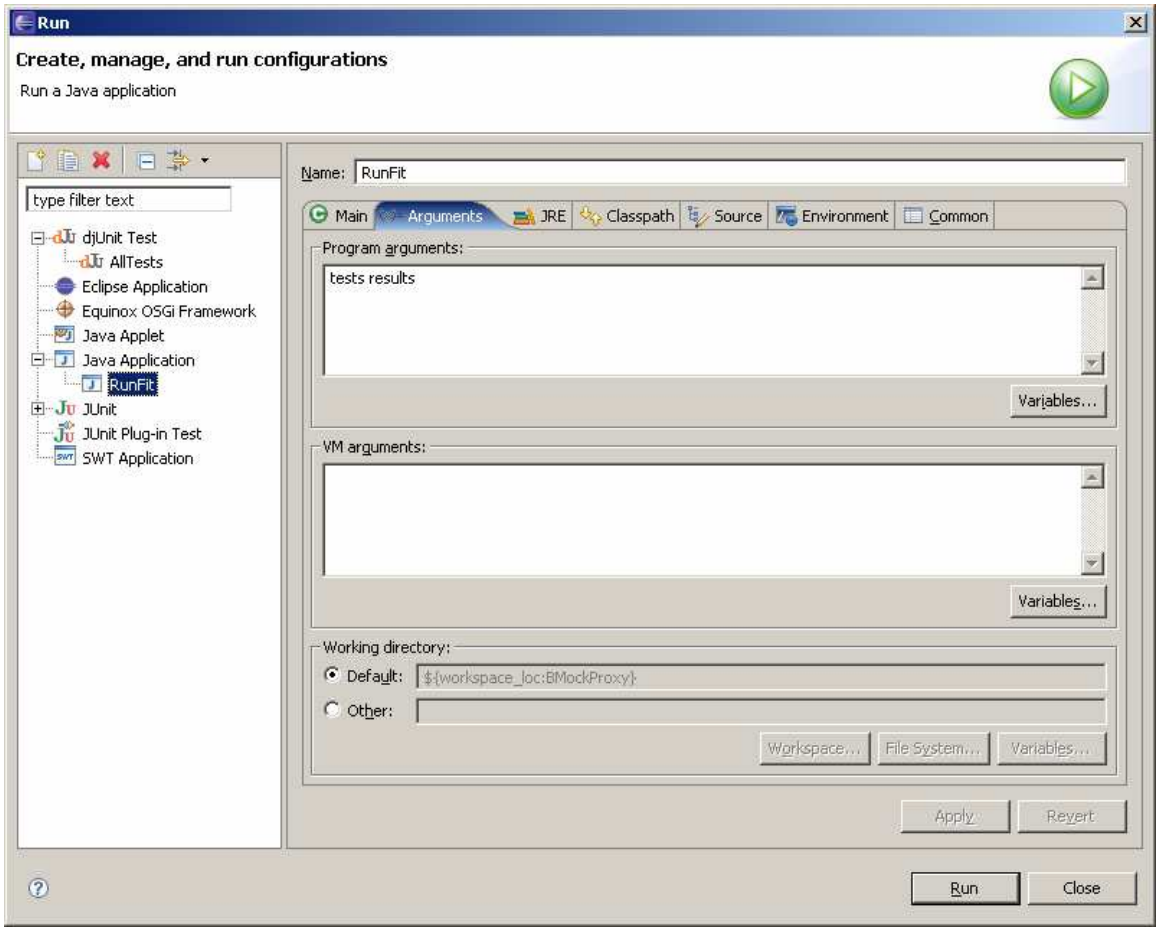
com.nds.bmock.console.SampleSetupFixture	
Field1	Field2
123	579

com.nds.bmock.console.EventJournal			
eventType	eventPayload	logRecord?	counter?
123	579, A	Log Record0	1
123	579, A	Log Record1	2
123	579, A	Log Record2	3
123	579, A	Log Record3	4
123	579, A	Log Record4	5
123	579, A	Log Record5	6
123	579, A	Log Record6	7
123	579, A	Log Record7	8
123	579, A	Log Record8	9
123	579, A	Log Record9	10

3. The FIT library comes with a handy folder runner which scans a particular file folder for test files and publishes results in another specified folder. This might be configured to be launched under Eclispe as follows:



4. The input and output folders are specified as follows:



5. In the example above the *SampleQueueConsumer* is so-called DoFixture class specified as follows:

```
import com.nds.bmock.console.Proxy;
import fitlibrary.DoFixture;

public class SampleQueueConsumer extends DoFixture {
      private transient       String     msg;
      private transient final Proxy      proxy;
      private transient final AmsLog     log;
      private transient final SampleSetupFixture consumerData;
      private transient final EventJournal eventJournal;

      public SampleQueueConsumer() throws Exception {
            super();
            this.proxy        = new Proxy(this);
            this.log          = new AmsLog(this);
            this.consumerData = new SampleSetupFixture(this);
            this.eventJournal = new EventJournal(this);
      }
```

6. Here the *Proxy* class comes from the com.nds.bmock.console.Proxy package, which is supplied within the bmockConsole.jar file within the \Boost\lib directory.

7. By default this class's **run()** method will look for an .EXE file, which is located in the current directory (where the FIT FolderRunner is running) and has the same name as the target class name (supplied in the Proxy constructor). In this particular case the Proxy class will look for a SampleQueueConsumer.exe file, which is supposed to be a normal **bmock** console mode application. All pipe wire and method call dispatching is performed automatically.
8. In case your bmock console application is located in another directory you can use the overloaded Proxy constructor:
   ```
   public Proxy(final Object fixture, final String dir)
   ```
   where dir parameter is the path to bmock console. The path can be absolute or relative. Remember to use double backslashes in the path string.
9. By convention the Proxy class will look for the same class and method name as specified in the console pipe text. For example, it will try to automatically map the "PscAdpater::getOpCode" string into the getOpCode() method of the PscAdapter class defined within the same package as the target class (supplied to the Proxy constructor). It will first try to obtain a pointer onto Psc object from the target class through the getPscAdapter() method. If not successful it will create a single copy of such object and will use it for all methods.
10. By convention all IN arguments are specified as is and OUT and IN_OUT arguments are specified as arrays. For instance:
    ```
    public Integer getOpCode(final Integer hCatalog
                            ,final Integer hProg
                            ,final Integer[] opCode)
    {
          opCode[0] = 1;
          return 1;
    }
    ```

11. Return value will be automatically printed into the pipe.
12. In addition the Proxy class provides a special support for event queues. For that purpose the Event queue class has to be passed as an argument to the Proxy class **start()** method, as follows:

    ```
    public void setUp() throws Exception {
          this.proxy.start(AmsQueue.class);
    }
    public void tearDown() throws Exception {
          this.proxy.stop();
    }
    ```

13. Running the proxy in setup() and stooping in tearDown() methods of DoFixture is a good practice and highly recommended.
14. In the example above events are submitted by using a form of ColumnFixture as follows:

    ```
    import fit.ColumnFixture;

    public class EventJournal extends ColumnFixture {
          private final transient SampleQueueConsumer consumer;
          public transient Integer eventType;
    ```

```
        public transient String  eventPayload;

        public EventJournal(final SampleQueueConsumer consumer) {
              super();
              this.consumer = consumer;
        }
        public String logRecord() throws Exception {
              this.consumer.sendEvent(
                    new AmsEvent(this.eventType
                                 ,this.eventPayload));
              return this.consumer.getLogMsg();
        }
  }
```

15. This ColumnFixture is in turn retrieved from the DoFixture class as follows (the FIT library convention):

```
  public EventJournal
        getComDotNdsDotBmockDotConsoleDotJunitDotEventJournal() {
        return this.eventJournal;
  }
```

16. Sending an event is simple as follows:

```
  public void sendEvent(final AmsEvent evt) throws Exception {
        this.proxy.put(evt);
  }
```

17. The AmsQueue class in example above has to be derived from the EventQueue class as follows:

```
public class AmsQueue extends EventQueue {
   public AmsQueue(Proxy proxy) {
         super(proxy);
   }

   public void getEvent(final Integer[] eventType
                              ,final String[] eventPayload)
                              throws InterruptedException
      {
         final AmsEvent event = (AmsEvent)super.get();
         eventType[0]    = event.getType();
         eventPayload[0] = event.getPayload();
   }
}
```

18. The **bmock** console infrastructure performs all required synchronization internally and ensures that only one event will be submitted and processed at once.
19. The **bmock** console infrastructure provides also supports “C” functions according to the following convention: a) any string in a form of “Module_Function” will be treated as a method “Function” of class “Module”. Any string in a form of “Function” will be treated as a method of the main DoFixture class, if any.

20. The **bmock** console infrastructure provides support for method with byte array arguments:
    - For methods with byte[] in arguments it automatically translates the hexadecimal input to bytes.
    - For methods with byte [] [] out arguments it automatically translates the out argument to hexadecimal value.
    - For methods with byte [] [] in_out arguments it supports the translation both ways.
21. The **bmock** console infrastructure also contains the class ByteArray which supports translation between Hexadecimal strings and byte arrays.
22. The class contains three constructors:
    ByteArray(**byte** [] array_)
    ByteArray()
    ByteArray(**final** String Hex)
    and these methods:

    int length()
    byte[] getBytes()
    String toStringHex()
    void hex2byte(**final** String Hex)

    In order to receive a hexadecimal string from a byte array one needs to use the *toStringHex* function as follows:

```
final byte [] byte_=new byte []{123,31};
final ByteArray array=new ByteArray(byte_);
String str=array.toStringHex();
assertEquals(str,"7B1F");
```

    There are two ways to receive a byte array from a hexadecimal string :
    1.

```
final String char_="7B1F";
final ByteArray array=new ByteArray();
array.hex2byte(char_);
final byte[] byte_=array.getBytes();
assertEquals(((Byte)byte_[0]).intValue(),123);
assertEquals(((Byte)byte_[1]).intValue(),31);
```

    2.

```
final String char_="7B1F";
final ByteArray array=new ByteArray(char_);
final byte[] byte_=array.getBytes();
assertEquals(((Byte)byte_[0]).intValue(),123);
assertEquals(((Byte)byte_[1]).intValue(),31);
```

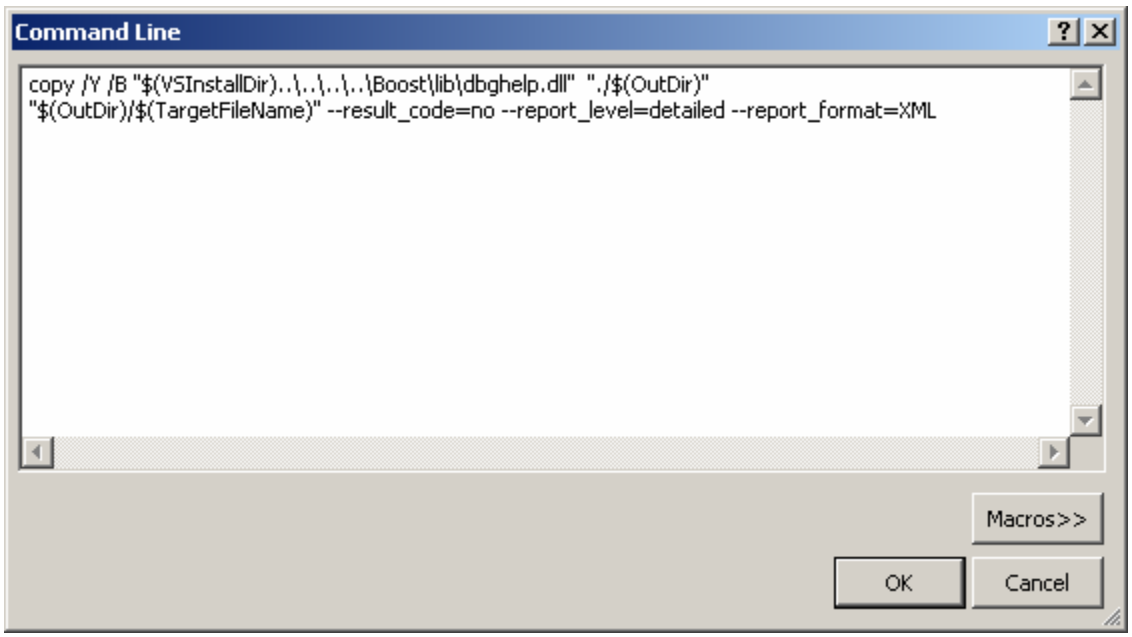
7 BullseyeCoverage

1. BullseyeCoverage is a code coverage analyzer for C++ and C that tells you how much of your source code was tested.

2. The tool supports both function coverage and condition coverage.
3. It integrates with Microsoft Visual Studio.
4. For more information see http://www.bullseye.com.

8 Known problems

23. RAW_MEM arguments with special, such as TLV, length treatment are not properly supported. Will be fixed in some future version. In the meantime specify them as RAW_MEM(IN_OUT,…) and try to play with the use some fixed length.
24. Keeping mock function declaration in a component header is not properly supported yet. Planned to be dealt with in some future version.
25. Under Windows 2000 the memory leak detector needs to be copied into the Debug directory. The easiest way to achieve this is to modify the corresponding post-build event as follows (a standard support will be provided in the next version):



26. The current version of the Boost.Threads library causes a memory leak of 32 bytes. A deep source code change of the Boost.Threads library is required in order to eliminate this memory leak. It might be done in some future version of the **bmock** library.