Boost Test Library

Usage in production components

Gennadiy Rozental
rogeeff@gmail.com

Boost Test

Part 1: Test and Testee … neighbors

Test and Testee … neighbors

Boost Test

- All about test organization – practical viewpoint.
- No best practice
- Each approach has Pro and Con points … sometimes the same
- Warm-up before we jump into more advanced staff

Test and Testee … neighbors

Boost Test

1. Living on the same block … common scenario

- Most common unit test configuration
- Unit test run in development environment
- Unit test built separately from library

Example:

```
…\my_library\
        \include\lib.hpp
        \src\lib.cpp
        \test\lib_test.cpp
        …
```

Test and Testee … neighbors

Boost Test

1. Living on the same block … common scenario

   Example: lib_test.cpp

```
#define BOOST_TEST_MODULE mytest
#include <boost/test/unit_test.hpp>
#include "../include/lib.hpp"
… // here are test cases
```

Test and Testee … neighbors

Boost Test

1. Living on the same block … common scenario

Pros:	Cons:
✓Clear separation of test and library source code	✓ Clear separation of test and library source code
✓ No affect of test code on library code at runtime	✓ Unit test – separate binary an require additional build step
✓ No affect on compilation time for both	✓ Test file – separate source code to maintain
✓ No affect on binary size	

Test and Testee … neighbors

Boost Test

2. Living under the same roof … but in parallel universe

- Good for active library development phase
- Good for small projects

Example:

```
…\my_library\
        \include\lib.hpp
        \src\lib.cpp
              \Makefile
        …
```

Test and Testee … neighbors

Boost Test

2. Living under the same roof … but in parallel universe

Example: lib.cpp

```
#include "lib.hpp"

// optionally #ifndef UNIT_TEST
… // here is library implementation
// optionally #endif
#ifdef UNIT_TEST
#define BOOST_TEST_MODULE mytest
#include <boost/test/unit_test.hpp>
… // here is library unit test
#endif
```

Test and Testee … neighbors

Boost Test

3. Living and coexisting in the same flat … but use different entrances

Usage:

1. While you develop build unit test target, which defines UNIT_TEST macro
2. When you are ready to release build library target, which does not define UNIT_TEST macro

Test and Testee … neighbors

Boost Test

2. Living under the same roof … but in parallel universe

Pros:	Cons:
✓ Tight integration of test and library source code	✓ No separation of test and library source code
✓No need for separate build step	✓ Need to build library code when building unit test (unless extra ifdef used)
✓ No affect of test code on library code at runtime, on compilation time and on binary size	✓ Changes in unit test lead to library rebuild

Test and Testee … neighbors

Boost Test

3. Living and coexisting in the same flat … but use different entrances

- What if you want to run unit test in production environment?
- You want to be sure unit test and library correspond to each other – use same binary

Example:

…\my_library\
\include\lib.hpp
\src\lib.cpp
…

Test and Testee … neighbors

Boost Test

3. Living and coexisting in the same flat … but use different entrances

Example: lib.cpp (only works for shared libraries)

```
#include "lib.hpp"
… // here is library implementation

#define BOOST_TEST_DYN_LINK
#include <boost/test/unit_test.hpp>
… // here is library unit test

// add init function
extern "C" { bool init_unit_test() { … } }
```

Test and Testee … neighbors

Boost Test

3. Living and coexisting in the same flat … but use different entrances

Usage:

1. Build you component as shared library (don't forget to add unit test init function)
2. Use this library in production as usual
3. To invoke unit test employ stand alone test runnier. For example one which comes with Boost.Test: concole_test_runner

console_test_runner --test your_lib.dll (or .so)

Test and Testee … neighbors

Boost Test

3. Living and coexisting in the same flat … but use different entrances

Pros:	Cons:
✓ Tight integration of test and library source code	✓ No separation of test and library source code
✓No need for separate build step	✓ Need to build library code when building unit test
✓ You can run the test in production environment based on binary which actually work there	✓ Changes in unit test lead to library rebuild ✓ Might affect production library size, performance

Boost Test

Part 2: Test? … What Test?

Test? … What Test?

Boost Test

- We'll cover usage of Boost.Test components for **non-test** purposes
- These include:
  - Program Execution monitor
  - Execution Monitor and debugger interfaces
  - Floating point comparison routines
  - Testing tools … reincarnated
  - Bonus: hidden utilities

Test? … What Test?

Boost Test

Program Execution Monitor
simple but powerful tool

- Relieves users from messy error detection and reporting duties
- Uniformly detects and reports several types of errors
- Not very configurable, but require only 2 lines of code to use

Test? … What Test?

Boost Test

Program execution monitor example:

```
extern int faulty_func();

int main( int, char *[] )
{
    return faulty_func();
}
```

Test? … What Test?

Boost Test

Program execution monitor example:

```
#include <boost/test/prg_exec_monitor.hpp>

extern int faulty_func();

int cpp_main( int, char *[] ) // note the name
{
    return faulty_func();
}
```

Test? … What Test?

Boost Test

Program execution monitor good for:

- Programs running unattended
- Legacy programs
- In general in any environment where uniform reporting of errors is necessary

Test? … What Test?

Boost Test

Program execution monitor may not be applicable if:

- You need more control over code execution
- You want more detailed error information
- For unit testing prefer Unit Test Framework

Test? … What Test?

Boost Test

Program execution monitor have 3 usage variants:

- Static library usage variant

```
#include <boost/test/prg_exec_monitor.hpp>
```

Test? … What Test?

Boost Test

Program execution monitor have 3 usage variants:

- Static library usage variant
- Single header usage variant

```
#include <boost/test/included/prg_exec_monitor.hpp>
```

Test? … What Test?

Boost Test

Program execution monitor have 3 usage variants:

- Static library usage variant
- Single header usage variant
- Shared library usage variant

```
#define BOOST_TEST_DYN_LINK
#include <boost/test/prg_exec_monitor.hpp>
```

(or in Makefile)

Test? … What Test?

Boost Test

Program execution power comes from using Execution Monitor – Boost.Test foundation component

Execution Monitor

Program Execution Monitor

Unit Test Framework

More information online

Test? … What Test?

Boost Test

Execution Monitor or…
not so simple life underground

- Generic low-level tool for controlled execution of any functions with a uniform error report
- Very versatile and allow fine tuning to match your needs

Test? … What Test?

Boost Test

Execution Monitor design rationale

- Uniformity
- Report as much as possible
- Flexibility
- Portability
- Stability (even if no memory)

Test? … What Test?

Boost Test

Execution Monitor basic example

```
int func() { throw "Some error"; }

int main()
{
    boost::execution_monitor ex_mon;
    try {
        ex_mon.execute( &func );
    }
    catch ( boost::execution_exception const& ex ) {
        std::cout << "Caught exception: “
                  << ex.what() << std::endl;
    }
    return 0;
}
```

Test? … What Test?

Boost Test

Execution Monitor interfaces

```
namespace boost {
class execution_exception {
  // an exception throw in case of errors
};
class execution_monitor {
  // primary interface
};
// 'non error' function abortion
struct execution_aborted {};
// FPE management interfaces
namespace fpe {…}
// debugger interface
namespace debug {…}
}
```

Test? … What Test?

Boost Test

Execution exception interface

```
namespace boost {
class execution_exception {
    enum error_code {…};
    struct location {
      const_string    m_file_name;
      size_t          m_line_num;
      const_string    m_function;
    };
    error_code      code() const;
    const_string    what() const;
    location const& where() const;
};
}
```

Test? … What Test?

Boost Test

Execution exception error codes

```
namespace boost {
class execution_exception {
  enum error_code {
    no_error,
    user_error,
    cpp_exception_error,
    system_error,
    timeout_error,
    user_fatal_error,
    system_fatal_error
  };
  …
};
}
```

Test? … What Test?

Boost Test

Execution Monitor properties

```
namespace boost {
class execution_monitor {
 readwrite_property<bool> p_catch_system_errors;

 readwrite_property<unsigned> p_detect_fp_exceptions;

 readwrite_property<int>  p_timeout;

 readwrite_property<bool> p_use_alt_stack;

 readwrite_property<bool> p_auto_start_dbg;
 …
};
}
```

Test? … What Test?

Boost Test

Execution Monitor properties

```
namespace boost {
class execution_monitor {
 template<typename ExceptionType, typename Tr>
 void  register_exception_translator(
    Tr const& tr, const_string tag );

 void erase_exception_translator( const_string tag );

 template<typename ExceptionType>
 void erase_exception_translator();
 …
};
}
```

Test? … What Test?

Boost Test

Execution Monitor: FPE management interfaces

```
namespace boost {
namespace fpe {
enum masks {
    BOOST_FPE_OFF, BOOST_FPE_DIVBYZERO,
    BOOST_FPE_INEXACT, BOOST_FPE_INVALID,
    BOOST_FPE_OVERFLOW, BOOST_FPE_UNDERFLOW,
    BOOST_FPE_ALL, BOOST_FPE_INV
};
unsigned BOOST_TEST_DECL enable( unsigned mask );
unsigned BOOST_TEST_DECL disable( unsigned mask );
}}}
```

Test? … What Test?

Boost Test

Execution Monitor: Debugger interfaces

```
namespace boost { namespace debug {
bool under_debugger();

void debugger_break();

bool attach_debugger( bool break_or_cont = true );

void detect_memory_leaks( bool on_off );

void break_memory_alloc( long mem_alloc_order_num );
}}
```

Test? … What Test?

Boost Test

Execution Monitor advanced examples

- 5 – catch_system_error
- 6 – detect_fp_exceptions
- 7 – auto_start_dbg
- 17 – timeout
- 8 – exception registration
- 9 – debugger interface

Test? … What Test?

Boost Test

Execution Monitor usage

```
// interface
#include <boost/test/execution_monitor.hpp>
#include <boost/test/debug.hpp>
…
// implementation
#include <boost/test/impl/execution_monitor.ipp>
#include <boost/test/impl/debug.ipp>
```

Test? … What Test?

Boost Test

Floating point comparison routines

- General purpose
- Have no dependencies
- Lightweight
- New version abstracted out even more

Test? … What Test?

Boost Test

Floating point comparison interfaces

```
namespace boost { namespace math { namespace fpc {
enum strength {
    FPC_STRONG, // "Very close
    FPC_WEAK    // "Close enough”
};

template<typename FPT>
class close_at_tolerance;

template<typename FPT>
class small_with_tolerance;

… more
}}}
```

Test? … What Test?

Boost Test

Floating point comparison interfaces

```
namespace boost { namespace math { namespace fpc {
template<typename FPT>
class close_at_tolerance {
  template<typename ToleranceType>
  explicit    close_at_tolerance(
      ToleranceType tolerance,
      fpc::strength fpc_strength = FPC_STRONG )

  bool operator()( FPT left, FPT right ) const;

  FPT             fraction_tolerance() const;
  fpc::strength strength() const;
};
}}}
```

Test? … What Test?

Boost Test

Floating point comparison interfaces

Why Template?

```
namespace boost { nam                    space fpc {
template<typename FPT>
class close_at_tolerance {
  template<typename ToleranceType>
  explicit    close_at_tolerance(
      ToleranceType tolerance,
      fpc::strength fpc_strength = FPC_STRONG )

  bool operator()( FPT left, FPT right ) const;

  FPT             fraction_tolerance() const;
  fpc::strength strength() const;
};
}}}
```

Test? … What Test?

Boost Test

Floating point: tolerance traits

```
namespace boost { namespace math { namespace fpc {
template<typename ToleranceType>
struct tolerance_traits {
    template<typename FPT>
    static ToleranceType
    actual_tolerance( FPT fraction_tolerance )

    template<typename FPT>
    static FPT
    fraction_tolerance( ToleranceType tolerance );
};
}}}
```

Test? … What Test?

Boost Test

Floating point: percent tolerance

```
namespace boost { namespace math { namespace fpc {
template<typename FPT>
struct percent_tolerance_t;

template<typename FPT>
inline percent_tolerance_t<FPT>
percent_tolerance( FPT v );
}}}
```

- Do we need tolerance in number of ULP?

Test? … What Test?

Boost Test

Floating point comparison interfaces

```
namespace boost { namespace math { namespace fpc {

template<typename FPT1, typename FPT2, typename
ToleranceType>
bool
is_close_to( FPT1 left, FPT2 right,
             ToleranceType tolerance );

}}}
```

Test? … What Test?

Boost Test

Floating point comparison interfaces

```
namespace boost { namespace math { namespace fpc {

template<typename FPT>
class small_with_tolerance {
public:
 explicit    small_with_tolerance( FPT tolerance );
 bool        operator()( FPT fpv ) const;
};

template<typename FPT>
bool
is_small( FPT fpv, FPT tolerance );
}}}
```

Test? … What Test?

Boost Test

Floating point comparison: open questions

- ULP based tolerance
- Tolerance defaults
- Dependency on boost::math headers

Test? … What Test?

Boost Test

Testing tools … reincarnated

- Rich set of interfaces to implement test assertions
- Macro based
- Detailed error messages
- More information online

Test? … What Test?

Boost Test

Testing tools … reincarnated

- Rich set of interfaces to implement test assertions
-
-
- online

Hmm … Where else I might have similar tasks??

Test? … What Test?

Boost Test

Testing tools … reincarnated

As production assertions!

Test? … What Test?

Boost Test

Testing tools as production assertions

- Existing solutions in plenty, but not as powerfull
- Need different semantic
- Need different implementation
- Can’t mix unit test and production code

Test? … What Test?

Boost Test

Testing tools as production assertions

To implement you only need to:

- define BOOST_TEST_PROD
- Implement macro BOOST_TEST_TOOL_IMPL
- #include <test_tools.hpp>

Test? … What Test?

Boost Test

Testing tools as production assertions

Experimental bundled implementation:

- Default tools semantic:
  - WARN – do nothing
  - CHECK – throw Boost.Exception
  - REQUIRE – use BOOST_ASSERT
- Efficient – no extra work if passed
- Configurable

Test? … What Test?

Boost Test

Testing tools as production assertions

Bundled implementation configuration is based on macros operating on string error description (define before prod_tools.hpp):

- BOOST_TEST_TOOL_REPORT_WARN_FAILURE
- BOOST_TEST_TOOL_REPORT_CHECK_FAILURE
- BOOST_TEST_TOOL_REPORT_REQUIRE_FAILURE

Test? … What Test?

Boost Test

Testing tools as production assertions

- 14 - basic example
- 15 - shows that no extra work is done
- 16 - configuration example

Note that you need to link with unit_test_framework static library

Test? … What Test?

Boost Test

Testing tools as production assertions

Open questions:

- Make sense?
- ODR violation?
- Best configuration interface?

Test? … What Test?

Boost Test

Bonus: hidden utilities

- Located under boost/test/utils
- Generally applicable
- No covered in docs
- Many have alternatives in boost

Test? … What Test?

Boost Test

Bonus: hidden utilities

- algorithm.hpp
- class_properties.hpp
- fixed_mapping.hpp
- lazy_ostream.hpp
- named_params.hpp
- nullstream.hpp
- rtti.hpp
- setcolor.hpp
- trivial_singleton.hpp
- xml_printer.hpp

18
21
19
20

- basic_cstring
- runtime
- iterator